VIETNAM NATIONAL UNIVERSITY – HOCHIMINH CITY
INTERNATIONAL UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

DIGITAL SYSTEM DESIGN
Final Project

DESIGN AND SIMULATION OF RISC-V PROCESOR USING VERILOG

SUBMITTED BY

NGUYEN THI THU QUYEN – EEEEIU21048

HO CHI MINH CITY, VIET NAM
2025

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS AND NOTATIONS

RGB: Red-Green-Blue

PPE: Personal Protective Equipment

VOC: Visual Object Classes

RPN: Region Proposal Network

EXIF: Exchangeable Image File Format

R-CNN: Regions with Convolutional Neural Networks

AMP: Automatic Mixed Precision

SGD: Stochastic Gradient Descent

mAP: mean Average Precision

ABSTRACT

This project presents the design and implementation of a RISC-V CPU architecture using Verilog Hardware Description Language (HDL), drawing inspiration from the pipelined structure of the MIPS processor. The designed processor supports 32-bit instruction words, utilizes two basic instruction formats, and features four 32-bit general-purpose registers. The architecture is entirely described in Verilog HDL, synthesized using Quartus Prime 24.1 Standard Edition, and functionally verified through testbenches to ensure logical correctness. Following successful simulation, the processor is deployed on the DE2-115 FPGA development board, which is equipped with an Altera Cyclone IV FPGA featuring 529 I/O pins and a 50 MHz onboard clock oscillator. The simulation results confirm the accuracy and stability of the design, highlighting its suitability for educational and embedded system applications on FPGA platforms.

CHAPTER I

INTRODUCTION

In the introduction, I will first discuss background and motivation. The second part presents objectives of the project.

1.1. Background and Motivation

The Reduced Instruction Set Computer (RISC) is a processor design philosophy that focuses on minimizing the number and complexity of instructions in the instruction set. It employs fixed-length instruction formats, simplifies control logic, and improves overall performance. In contrast, Complex Instruction Set Computer (CISC) architectures utilize more complex instructions, various addressing modes, and require more intricate control units. Due to its simpler architecture, a RISC processor consumes less silicon area, enabling the integration of additional on-chip peripherals such as timers, interrupt controllers, and I/O modules—aligning with the growing trend of system-on-a-chip (SoC) solutions in the embedded systems market.

Simultaneously, Field Programmable Gate Arrays (FPGAs) have become increasingly popular due to their reprogram ability, lower cost compared to ASICs, and suitability for educational and research applications. In the process of studying computer architecture, knowledge is often gained primarily through software-based simulations, which can limit exposure to real hardware implementation. Therefore, integrating hardware design on FPGA platforms allows the practitioner to gain deeper insight into the relationship between hardware and software, the instruction execution process, and the internal operations of a functioning processor.

1.2. Objectives of the Project

The objective of this project is to design and simplify a simplified RISC-V RV32I processor using Verilog HDL and implement it on the DE2-115 FPGA development board.

Specific goals include:

- To study the RISC-V RV32I instruction set architecture and understand the core principles of RISC-based processors.
- To design and implement the main functional blocks, including instruction fetch, decode, execute, memory access, and write-back stages.
- To simulate and verify the correctness of each module using testbenches.

- To synthesize the processor design and deploy it onto the FPGA using Quartus Prime software.
- To strengthen understanding of hardware-software integration and digital system design through practical implementation.

CHAPTER II

THEORETICAL BACKGROUND

2.1. RISC-V Instruction Set Architecture (RV32I)

The RISC-V Instruction Set Architecture (ISA) is an open-standard, royalty-free architecture based on the principles of Reduced Instruction Set Computing (RISC). It was designed to be simple, modular, and extensible, making it suitable for both academic and industrial applications. Among its various base instruction sets, RV32I refers to the 32-bit integer base ISA, which forms the foundation of many RISC-V processors.

RV32I follows a load-store architecture, where memory is only accessed via specific instructions (e.g., LOAD, STORE), and all arithmetic or logical operations are performed on register-based operands. This simplifies the datapath and control logic, enabling faster execution and easier hardware implementation.

A key characteristic of RV32I is its use of fixed-length 32-bit instructions, which allows straightforward instruction decoding and supports efficient pipelined execution. The instruction formats are standardized and typically fall into one of the following types:

- R-type (Register): used for arithmetic and logic operations between registers
- I-type (Immediate): used for operations with immediate values and loads
- S-type: for store instructions
- B-type: for conditional branches
- U-type: for upper immediate instructions
- J-type: for jump instructions

Each instruction operates on 32 general-purpose registers, labeled x0 to x31, where x0 is hardwired to zero shown in figure X. The ISA supports common integer operations such as ADD, SUB, AND, OR, XOR, as well as comparison and shift instructions.

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0−2	x5−7	Temporary registers
s0/fp	x8	Saved register/Frame pointer
s1	x9	Saved register
a0−1	x10−11	Function arguments/Return values
a2−7	x12−17	Function arguments
s2−11	x18−27	Saved registers
t3−6	x28−31	Temporary registers

Figure 1: *RISC-V register set*

RV32I is designed to be minimal yet sufficient to support C/C++ compilers, linkers, assemblers, and operating systems. It avoids architectural complexities found in CISC architectures, such as variable-length instructions or microcode-based execution. Notably, RISC-V also eliminates features like branch delay slots, making control flow more predictable in pipeline design.

The modularity of RISC-V allows RV32I to be extended through instruction set extensions, categorized as:

- Standard extensions (e.g., M for multiplication/division, F for floating point)
- Reserved extensions for future use
- Custom extensions defined by hardware designers for application-specific needs

In this project, RV32I serves as the core instruction set for designing and implementing a simplified RISC-V processor. Its compact design and clear execution model make it an ideal choice for FPGA-based processor development, especially in educational or research-oriented environments.

2.2. Verilog Hardware Description Language

Verilog is a widely used hardware description language for modeling, simulating, and designing digital circuits. Standardized as IEEE 1364, it enables designers to describe hardware behavior and structure at multiple abstraction levels, including behavioral, register-transfer level (RTL), and gate-level.

Its C-like syntax makes Verilog accessible to those with programming backgrounds, while its concurrent execution model aligns with real-world hardware operation.

Key advantages of Verilog in digital design:

- Modular design: Circuits are structured as reusable modules with defined inputs and outputs.
- Simulation and verification: Testbenches allow functional testing before hardware synthesis.
- Synthesis-ready: Code can be synthesized into FPGA or ASIC hardware using tools like Quartus, Vivado, or ModelSim.
- Parallelism: Constructs like always, assign, and initial represent hardware events happening in parallel.

CHAPTER III

SYSTEM DESIGN AND ARCHITECTURE

3.1. System Overview

3.1.1. Single Cycle

Single-cycle processors execute one instruction per clock cycle, making them the simplest type of processor to design. The data path is constructed by interconnecting components like the ALU, instruction memory, program counter, multiplexers, register file, data memory, adders, and sign extension module, as shown in Figure 4.3.10. A control unit is incorporated to identify the current instruction and generate appropriate control signals to orchestrate the operation of these components

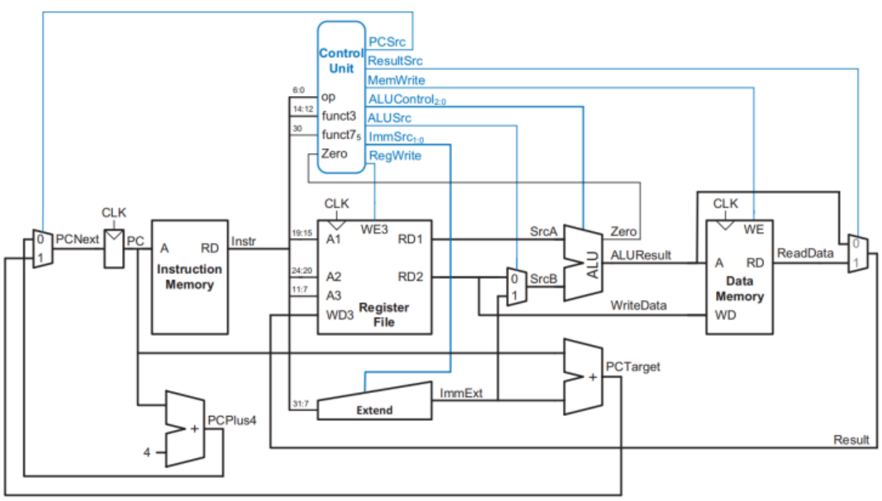


Figure 2: *Single cycle Datapath*

3.1.2. Pipeline

Pipelining enhances a processor's throughput by dividing the single-cycle processor into five stages: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM), and Writeback (WB). This allows up to five instructions to be processed simultaneously, one in each stage. In the IF stage, instructions are retrieved from the instruction memory. The ID stage decodes the instruction, reads source operands from the register file, and generates control signals. The EX stage performs ALU computations, the MEM stage handles data memory read/write operations, and the WB stage writes results back to the register file.

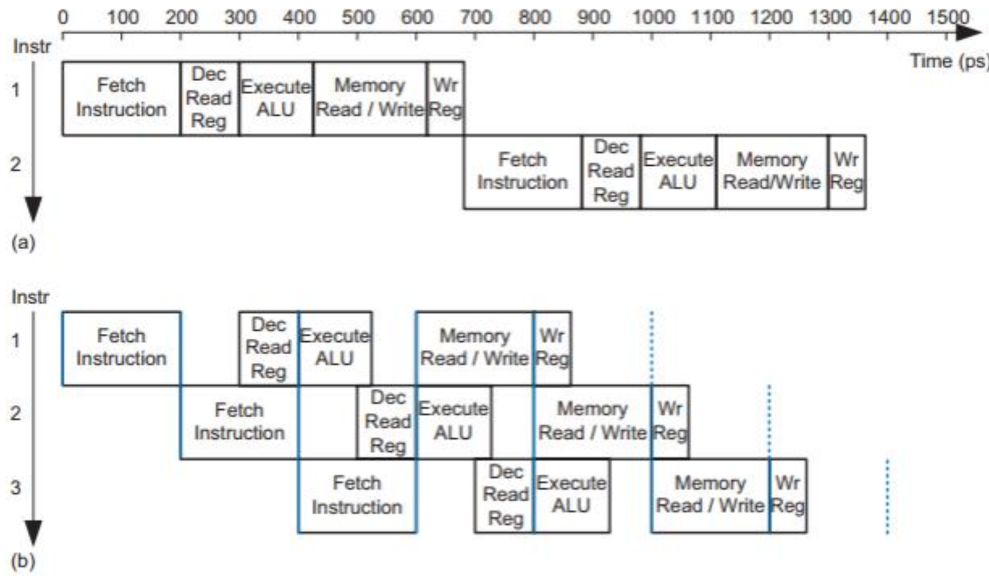


Figure 3: *Timing diagrams: a/ single cycle processor and b/ pipeline processor*

Since one cycle was split into five smaller parts, each cycle can only execute part of the instruction, but the processor can now execute five instructions in one cycle. The overall logic of each stage is only one-fifth, so the clock speed is about five times faster. Therefore, ideally, the latency of each instruction does not change, but the throughput is increased by a factor of 5. Nonetheless, pipelines offer tremendous benefits at low cost enough to pipeline all the latest high-performance microprocessors. Some processors have a 6-stage pipeline, with additional stages called publishing stages. Stages are designed to perform multiple issuance techniques that allow a single stage to execute multiple instructions. Theoretically, the throughput of these processors will be doubled, tripled, or more. The pipeline processor is created by adding four pipeline registers between each stage, as shown in the following figure.

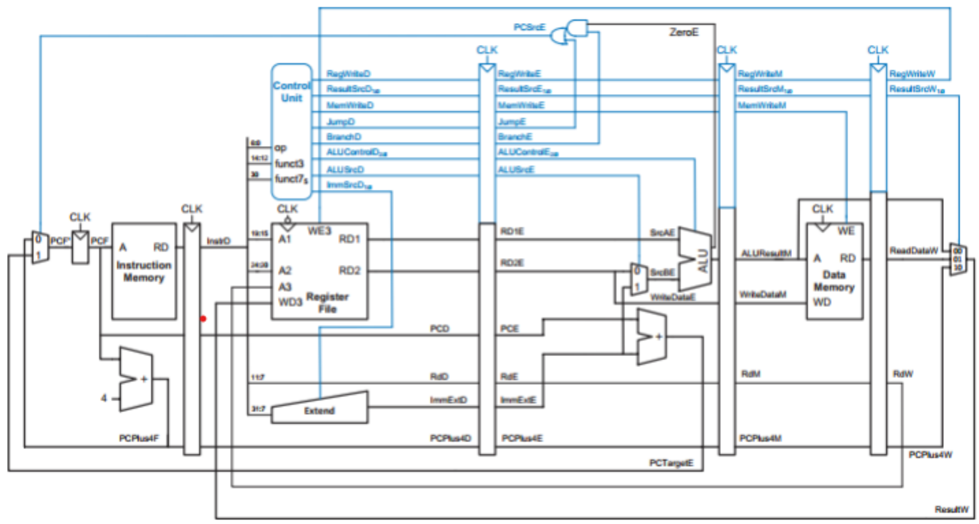


Figure 4: *RISC V processor with pipeline register*

3.1.3. Hazard

Pipelining enhances processor performance by executing up to five instructions concurrently across five stages, but it introduces hazards when a subsequent instruction requires the result of a prior instruction that is not yet complete. These hazards—structural, data, and control—are managed by a hazard unit using stall, flush, and forwarding (bypassing) techniques. The hazard unit monitors register sources (Rs1, Rs2), destination (Rd), the current program counter (PC), and control signals from the control unit to identify and resolve hazards.

Structural hazards occur when multiple instructions compete for the same data path resource simultaneously. One common issue arises with the register file, which is accessed during both the Writeback (WB) and Memory (MEM) stages, causing conflicts if read and write operations occur in the same cycle. This is resolved by configuring the register file to write on the negative clock edge, allowing both operations within one cycle. Another potential structural hazard involves using a single memory for both instructions and data, but this design avoids the issue by employing separate memories, eliminating the need for hazard unit intervention.

Data hazards arise when an instruction attempts to read a register before a previous instruction has written its result. A basic solution is to insert two NOP instructions between conflicting instructions, but this introduces a two-cycle latency, reducing performance. A more efficient approach is forwarding, where the hazard unit monitors Rs1 and Rs2 in the Execution (EX) stage and Rd and write-enable signals in the MEM and WB stages. When a hazard is detected, the required data is forwarded directly to the EX-stage, by passing the register file write and avoiding a two-cycle delay. This requires two additional multiplexers before ALU. Figures 15 and 16 illustrate the performance comparison between using NOP instructions and forwarding, while Figure 17 shows the processor with added forwarding components.

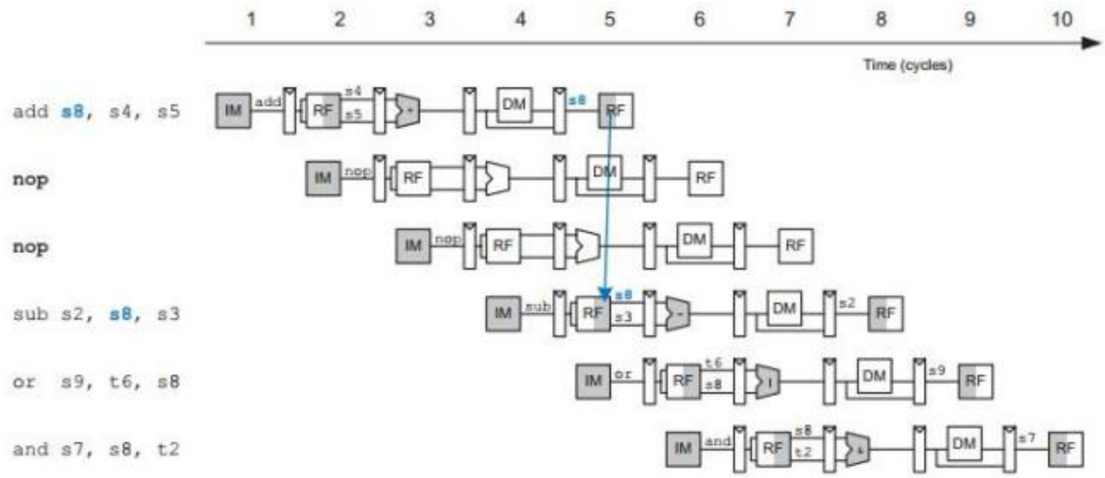


Figure 5: *Solving data hazard with NOP instruction*

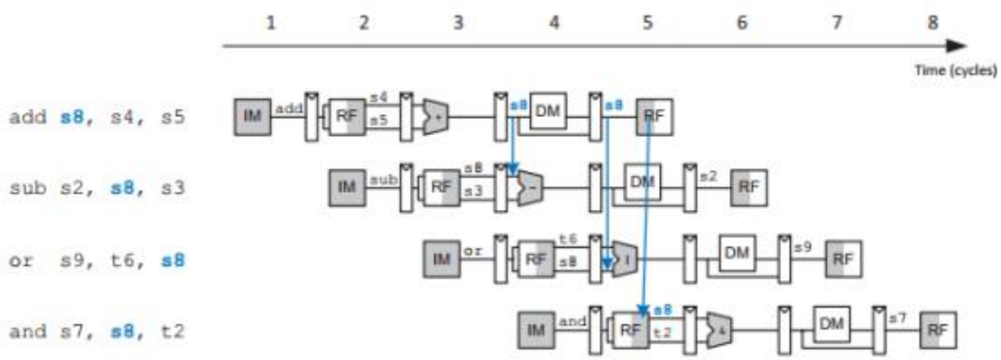


Figure 6: *Solving data hazard with forwarding*

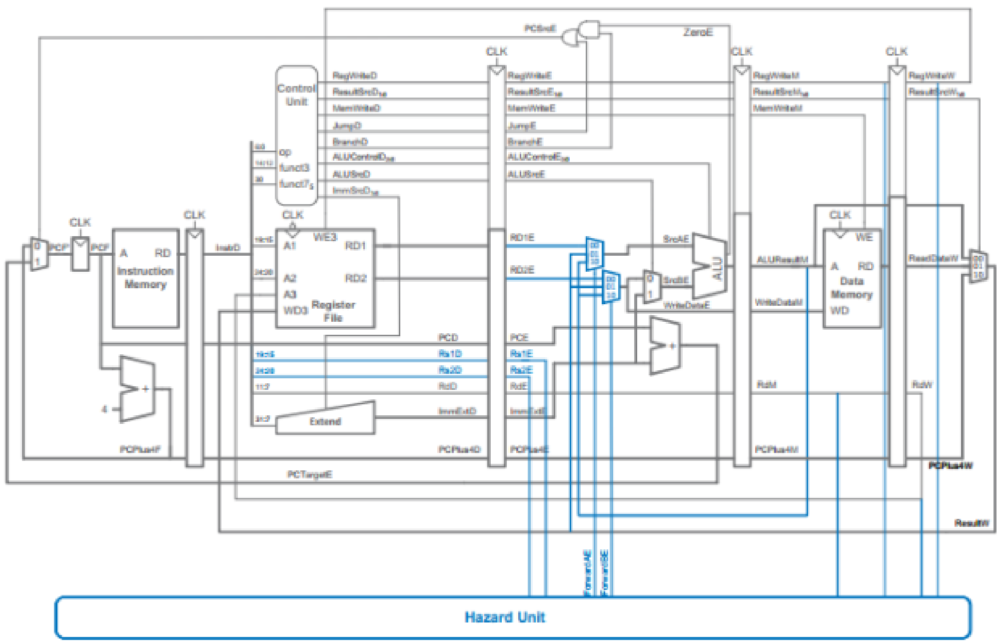


Figure 7: *RISC-V processor with forwarding technique*

Comparing Figures 15 and 16, it is evident that inserting NOP instructions to resolve data hazards reduces processor performance by increasing power consumption and execution time due to redundant instructions. Forwarding, or bypassing, is a more effective technique to mitigate these issues by avoiding unnecessary delays. However, forwarding cannot address all data hazards, particularly those caused by the load-word (lw) instruction, which retrieves data during the Memory (MEM) stage. Since forwarding multiplexers operate in the Execution (EX) stage, they cannot resolve lw-related hazards, resulting in a two-cycle delay. To handle this, the stall technique is employed, pausing the pipeline until the required data is available. This introduces a "bubble," similar to an NOP instruction, by nullifying the EX-stage control signals during an Instruction Decode (ID) stage stall, ensuring no architectural state changes. Stalling involves disabling the pipeline register to freeze stage inputs, with all prior stages also stalled to prevent instruction loss, and the subsequent register flushed to avoid incorrect data propagation. Stalls negatively impact performance and should be minimized.

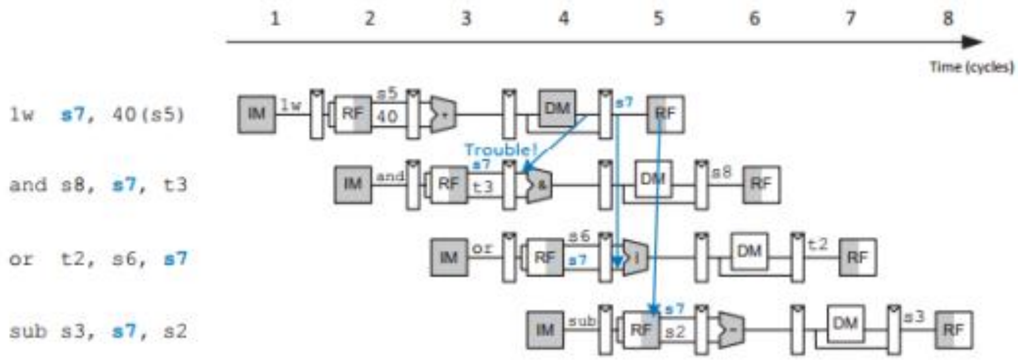


Figure 8: *LW data hazard*

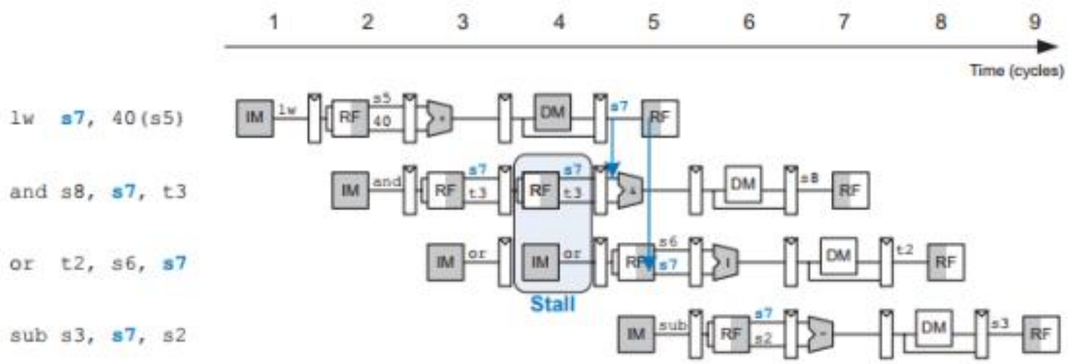


Figure 9*: Using stall technique to solve lw data hazard*

As illustrated in Figure 18, the load-word (lw) instruction creates a data hazard for the subsequent instruction, which relies on the loaded value to perform its computation. The following instruction, such as an AND operation, must wait until the lw instruction completes reading data and writes it to the target register (e.g., register 7, also referred to as register 23). This requires stalling the pipeline at the Instruction Decode (ID) stage for two cycles, resulting in the insertion of two NOP instructions due to a flush, which negatively impacts performance. Fortunately, the processor can utilize forwarding to mitigate this issue. By forwarding the data read by the lw instruction from the Memory (MEM) stage, the stall is reduced to a single cycle, as shown in Figure 19, leading to only one NOP instruction and improving overall performance.

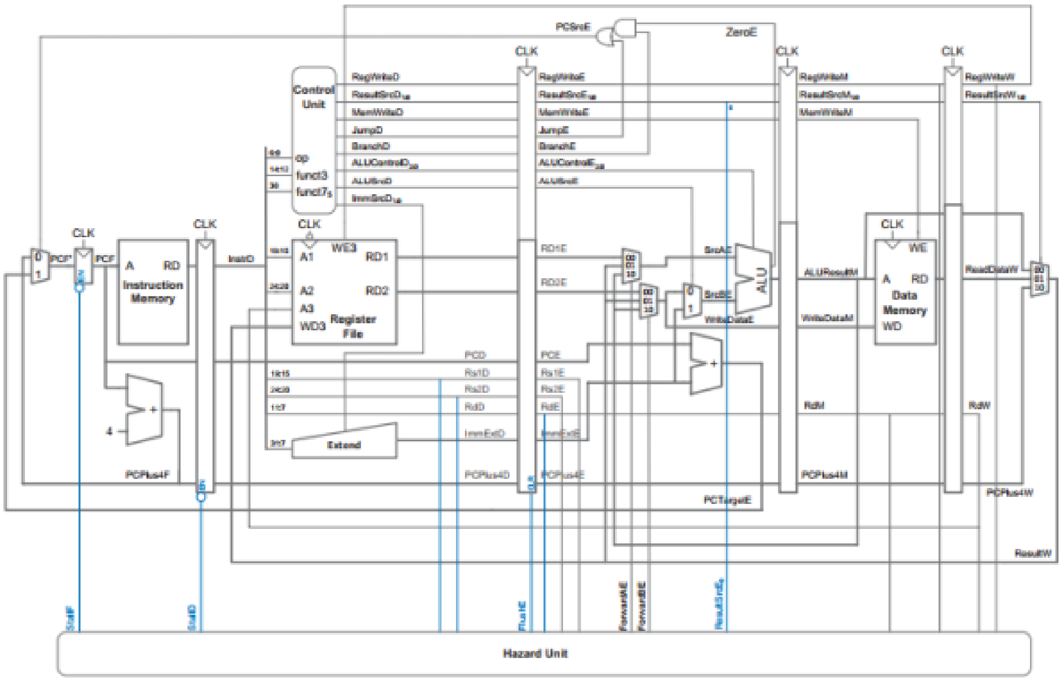


Figure 10: *RISC-V processor with stall technique*

Control hazards arise from jump or branch instructions, where the next program counter (PC) value is determined by a computation in the Execution (EX) stage, rather than the usual PC + 4. To address this, the hazard unit monitors the PCSrc signal in the EX stage and stalls the pipeline until the computation is complete, enabling a flush signal to clear the Instruction Decode (ID) stage outputs. As shown in Figure 21, a branch instruction triggers the flushing of the next two instructions, resulting in two additional NOP instructions, which degrades processor performance. Unlike structural and data hazards, control hazards cannot be mitigated by forwarding, so frequent occurrences significantly reduce throughput. Therefore, minimizing control hazards is critical to maintaining optimal performance.

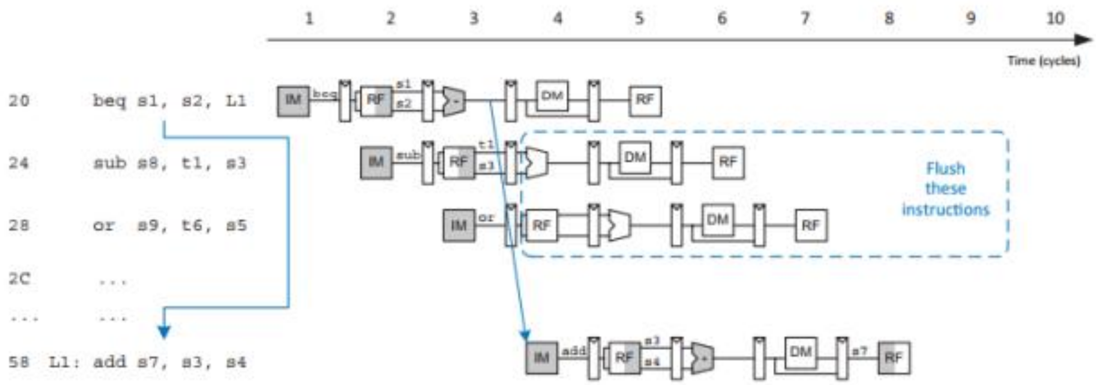


Figure 11: *Control hazard*

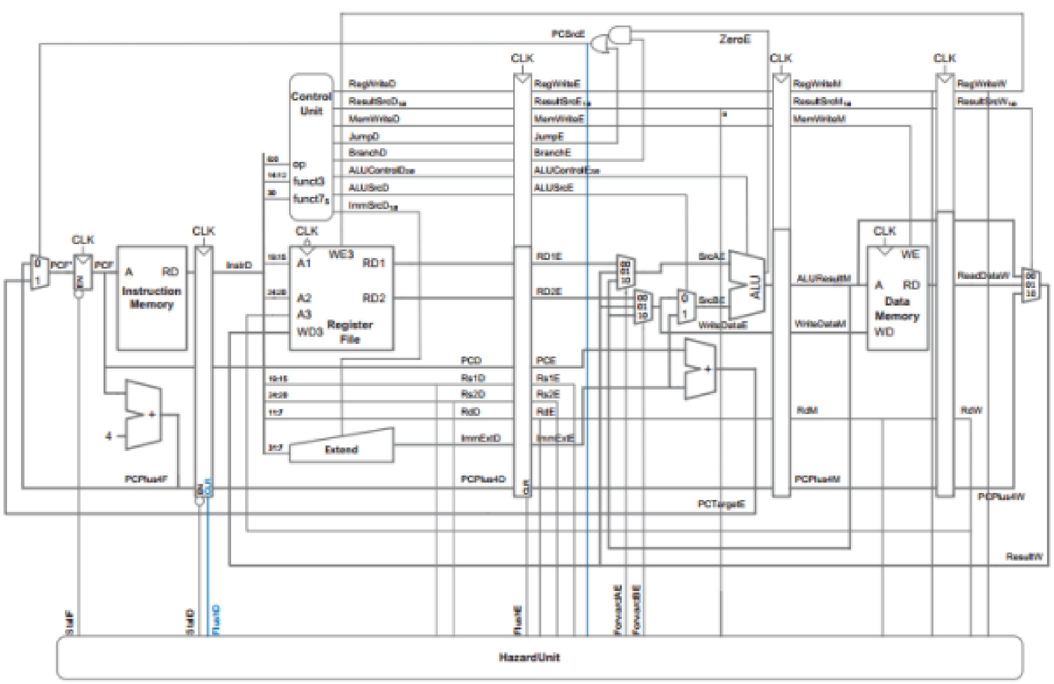


Figure 12: *Complete RISC-V processor*

3.2. RISC-V Component Design

3.2.1. ALU

The ALU is a vital component that performs a range of mathematical and logical operations, acting as the central processing element of the system. It operates based on a 2-bit control signal that dictates the specific operation to be executed. In addition to its primary output, the ALU generates flags that provide details about the result, including the N (negative), Z (zero), C (carry), and V (overflow) flags.

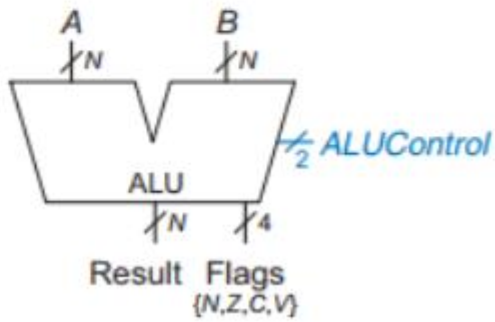


Figure 13: *N-bit ALU with Flag output and 2-bit ALU control signal*

ALU accepts two 32-bit inputs, which are processed for various computations or comparisons based on the ALU control signal. In this project, the ALU supports 16 distinct operations, including calculations and comparisons, plus one no-operation case that performs arithmetic addition by default. Implementing integer and logic calculations is straightforward, but the challenge lies in designing a fully functional comparator to handle all B-format instructions. To achieve this, bit 0 of the ALU output port is utilized to indicate comparison results: a 0 signifies a false comparison, while a 1 indicates a true comparison.

The ALU code encompasses 16 cases, each executing a specific computation, with 6 cases for comparisons and 10 for calculations. A dedicated "function" block within the ALU handles signed comparisons. Table 6 details the ALU module's port declaration. The ALU operates without a clock signal, responding solely to changes in its inputs.

Table 1: *ALU port declaration*

Name	Port type	Bit width	Description
alu_a_i	Input	32	- First term of the ALU.
alu_b_i	Input	32	- The second term of the ALU.
alu_op_i	Input	5	- Select signal to control the behavior of the ALU module. - The are 16 types of operation in this ALU, the alu_op_i signal should be ranging between 5'd1 to 5'd16, any other value is assigned to default state, which perform arithmetic addition between two inputs.
alu_op_o	Output	32	- Whenever any of its inputs change, this 32-bit port gets to be updated. - Bit 0 of this output signal is read whenever conditional jump instructions are in used.

For testing, the ALU's inputs are randomly generated one thousand times, triggered on the rising clock edge. Although the ALU itself is not clock-driven, the test environment uses a clock signal to control the timing of new random test generations. The alu_op_i input is defined as a "randc" (random cyclic) type, unlike other inputs which are "rand" (random). The "randc" type ensures that every possible value within the signal's range is generated exactly once, guaranteeing that all 16 computation cases are tested. A constraint is applied to limit the randomization of the operation code, preventing unnecessary test cases. The ALU is considered verified when all calculations and comparisons produce correct outputs for all test cases.

3.2.2. Instruction Memory

The instruction memory (IM) stores a program's instructions, which the processor retrieves by supplying an address to the IM for execution. It features a single output port. The instruction pointer (IP) uses a 32-bit value from the program counter (PC) to fetch the instruction data at the specified address, which is then output as read data.

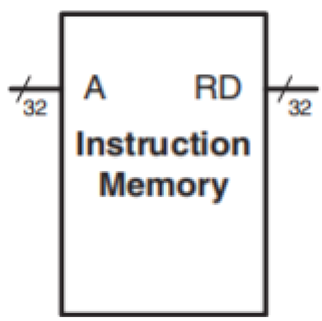


Figure 14: *Instruction Memory Schematic*

The Instruction Memory retrieves a 32-bit input address from the PC and outputs the corresponding 32-bit instruction. With a 32-bit address, it can theoretically hold up to 4,294,967,296 memory slots, equivalent to roughly 536,870,912 bytes. However, for the purposes of this thesis, which focuses on design and simulation, such a large memory capacity is unnecessary. Therefore, the IM is configured with only 1024 memory slots. Table 2 provides the port declaration, detailing the bit width and functions of all inputs and outputs for the instruction memory.

Table 2: *Instruction memory port explanation*

Name	Port type	Bit width	Description

read_address	Input	32	- Instruction addresses input. - Each address is corresponded to an instruction that is embedded by the instruction memory
instruction	Output	32	- The instruction output of this module. - Read the corresponding instruction based on the address input. - Every time the address input changes, a new instruction is updated.

To effectively test the IM module, a hybrid approach combining random testing and test vector methods is required. The address input should be randomly generated on each rising clock edge, and the DUT's output must be verified against the test vector to ensure it matches the expected results for the given address. To facilitate this, the test environment's scoreboard must access the test vector for comparison.

3.2.3. Program Counter

The program counter (PC) is a 32-bit binary counter that facilitates sequential instruction execution. It includes clock and reset inputs and a 32-bit output. A reset signal sets the output to zero, while each rising clock edge increments the PC by one, pointing to the current instruction's address in the RISC-V core.

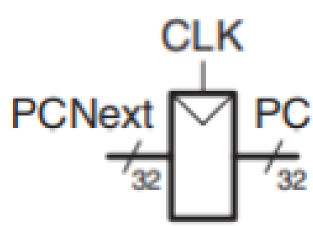


Figure 15: *Program Counter Schematic*

The program counter (PC) module functions by outputting a value that matches its input on every rising edge of the clock signal. Since the PC operation depends on the clock cycle, it is necessary to include a sensitivity list within the module using an "always" block. This "always" block allows the module to monitor specific variables and respond immediately whenever those variables change.

To clarify the code structure, Table 1 details all the ports of the program counter module, specifying each port's type, bit width, and a description of its role within the module**.**

Table 3: *PC module port description*

Name	Port type	Bit width	Description
clk	Input	1	- Clock pulse input that drives the behavior of the PC module. - It is the time reference of the PC module. - When clock pulse is on its rising edge, the PC module performs its function and update the output signal.
reset	Input	1	- Asynchronous reset signal of the PC module. - When reset signal is HIGH, the program counter output returns 0 despite what it input is.
stallF	Input	1	- Asynchronous signal of the program counter module - When the StallF is HIGH and reset signal is LOW, the program counter returns the output of the previous cycle
PC_in	Input	32	- The PC_in read the 32-bit input data.
PC_out	Output	32	- On rising edge of clock, if both StallF and reset signal are LOW, PC_out returns the PC_in data. - Else, PC_out returns the value with respect to the two cases as mentioned above.

The primary purpose of testing the designed module is to verify whether it performs its intended function correctly. For the program counter module, this means checking if the output value accurately reflects the input value, held for one clock cycle. Upon the next rising clock edge, the PC should output the input data from the previous cycle. To validate the module's functionality, one thousand random test cases were executed.

3.2.4. Multiplexer

Multiplexers select values for computation or load/store operations. This project employs two types: a 32-bit 4-to-1 multiplexer and a 32-bit 2-to-1 multiplexer. The 4-to-1 multiplexer has four 32-bit inputs and one 32-bit output, controlled by a 2-bit select signal. The 2-to-1 multiplexer has two 32-bit inputs and one 32-bit output, controlled by a 1-bit select signal.

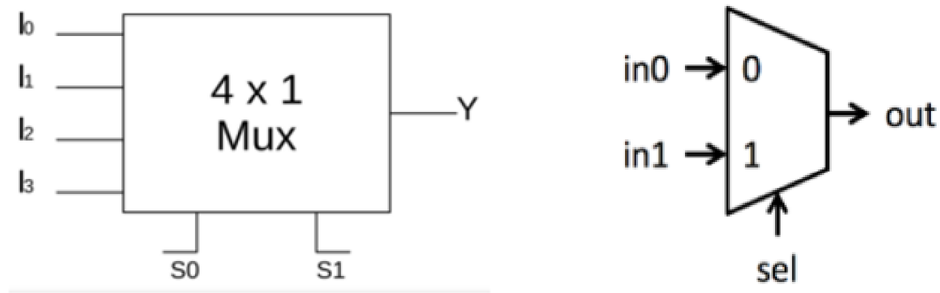


Figure 16: *4 to 1 mux (left) and 2 to 1 mux block (right)*

3.2.5. Register File

The register file, typically implemented as a compact multiport SRAM array, stores temporary variables in digital systems. In RISC-V, register 0 is fixed at zero. The register file has two 5-bit address read ports and one 5-bit address write port, allowing simultaneous reading of two 32-bit data values and writing of one. A control signal enables write operations, synchronized by a clock signal.

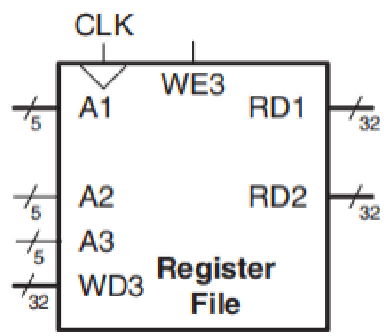


Figure 17: *Register file Schematic*

The register file adheres to two key rules: firstly, it contains 32 registers, each 32 bits wide; secondly, register 0 is always set to zero. This module requires a clock signal input, as the pipeline design mandates that the register file reads on the positive clock edge and writes on the negative clock edge to prevent structural hazards. Table 4 details the port declaration for the RF module, which includes six inputs and two outputs. The RF uses 5-bit addresses, limiting it to 32 memory slots, as a 5-bit address can only index up to 32 locations.

Table 4: *Register File port declaration*

Name	Port type	Bit width	Description
clk	Input	1	- Clock pulse which drives the behavior of register file module.

			- On the rising edge of clock, the register file performs read sequence, which update the two inputs of the register file. - On the negative edge of clock, the register file can start the write sequence, which write a 32-bit data into the internal memory.
read_addr_1	Input	5	- A 5-bit address let the register file know which memory to be read.
read_addr_2	Input	5	- A 5-bit address let the register file know which memory to be read.
write_addr	Input	5	- A 5-bit address let the register file know which memory to be write.
RegWrite	Input	1	- An asynchronous signal notifies the register file when to enable the write sequence. - When the signal is HIGH, the write sequence is enabled, which allow register file to start writing on negative edge clock.
write_data	Input	32	- A 32-bit data that is written into the memory of register file when RegWrite is HIGH. - The memory slot to be written is corresponded to the write_addr signal.
read_data_1	Output	32	- The 32-bit output that reads the memory slot which has the address corresponding to the read_addr_1.
read_data_2	Output	32	- The 32-bit output that reads the memory slot which has the address corresponding to the read_addr_2.

For verification, the input ports—read_addr_1, read_addr_2, write_data, write_addr, and RegWrite—are randomized. Read and write operations are enabled randomly to test for conflicts between these functions. The module is deemed successful when it correctly reads data from or writes data to the specified addresses.

3.2.6. Data Memory

The data memory features one 32-bit write address port, one 32-bit read address port, and one 32-bit input port for writing data. When the write enable (WE) signal is active (1), data is written to the address provided by the ALU on each rising clock edge. Otherwise, the memory reads data from the specified address to its output. Designed for load and store instructions, the data memory has a single output port, unlike the register file.

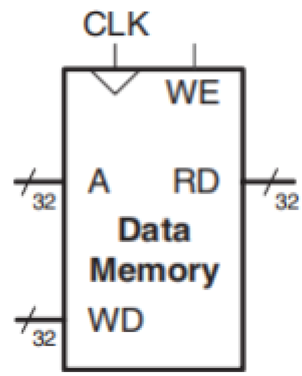


Figure 18: *Data Memory Blocks*

The Data Memory, the second memory core, supports both read and write operations. It features one 32-bit write input, one 32-bit read output, a MemWrite signal to enable the store function, and a 32-bit address input. In this processor, the Data Memory is configured with 64 memory slots and performs write operations on the positive clock edge. However, read operations occur asynchronously, as they do not need to be synchronized with the system. Table 7 provides the port details for the DM module. Although a 32-bit address allows for up to 4,294,967,295 memory slots, this project, intended for educational purposes, requires only 64 memory slots, making a larger capacity unnecessary.

Table 5: *Data memory port declaration*

Name	Port type	Bit width	Description
clk	Input	1	- Time control signal for DM module. - Allow data memory module to write on every rising edge of clock.
MemWrite	Input	1	- 1-bit signal which enable the write sequence of data memory.
addr	Input	32	- A 32-bit input that inform the DM module know which memory slot to access for read or write.
write_data	Input	32	- A 32-bit input that carries data to be written into DM module.
read_data	Output	32	- The read data that is continuously driven at any given time.

The address port is generated randomly and has a constraint which only allow generated data from 0 to 63. Other data to be random include write_data and MemWrite. One thousand test case is conducted to test for the validity of this module.

3.2.7. Adder

Adders perform additional operations within the core. Two adders are used: one for the program counter, which adds a 32-bit immediate value of 4 to the current instruction address to compute the next address, and another for branch and jump instructions, which adds the current address to an immediate value for branch, jump-and-link, or jump-and-link-register instructions.

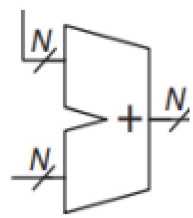


Figure 19: *Adder block*

The Adder module accepts two 32-bit inputs and performs their addition. Due to its straightforward functionality, this module is implemented using data flow coding. For verification, both inputs are randomly generated. The module is considered successfully verified when it consistently produces the correct sum of the two inputs whenever they change. Table 3 outlines the port details of the added module, including two inputs and one output.

Table 6: *Adder module port declaration*

Name	Port type	Bit width	Description
a	Input	32	- First term of the addition.
b	Input	32	- First term of the addition.
Out	Output	32	- First term of the addition.

3.2.8. Sign Extend

The sign extension module extends an N-bit immediate value to 32 bits, matching the ALU's bit width. It takes an N-bit immediate and a control signal as inputs, replicating the most significant bit (MSB) of the immediate until it reaches 32 bits. The control signal varies based on the instruction, resulting in different immediate values for each instruction type.

Figure 20: *Sign extend block*

The Sign Extend module takes a 25-bit input and extends it based on specific cases. Depending on the instruction type, the immediate value's arrangement varies. To handle this, a control signal called ImmSrc and a "switch case" mechanism are implemented to enable the sign extend module to process different instruction types appropriately. Figure 49 provides details on the immediate value arrangement for each instruction format.

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate
11	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-bit signed immediate

Figure 21: *Sign extend cases from reference*

According to Figure 49, the Sign Extend module initially handles four cases. However, an issue arises with the I-type shift instruction, which uses bits 24 to 20 of the instruction as the shift amount, conflicting with the standard I-format sign extension. Additionally, Figure 49 does not account for LUI/AUIPC instructions. To address these, two additional cases are introduced: one for the I-type shift instruction and another for LUI/AUIPC instructions. The formats for these two extra cases are presented in Figure 50.

ImmSrc	ImmExt	Type	Description
100	{Imm[24:5], 12'b0}	U	12-bit lower extend
101	{27'd0, Imm[17:13]}	Shift	27-bit upper extend

Figure 22: *Extra sign extends cases*

The Sign Extend module is designed with two inputs and one output. As the immediate value varies across different instruction types, the entire 32-bit instruction, excluding the 7-bit opcode, is used as the 25-bit input signal. This signal is then analyzed, segmented into smaller parts, and recombined based on the specific instructions being executed in the current program.

Table 7: *Sign extend module port declaration*

Name	Port type	Bit width	Description
Imm	Input	25	- The input signal for extension. - This signal when read by the sign extend module, is extended based on the cases in Figure 50.
ImmSrc	Input	3	- 3-bit input to select which extension case to perform. Hence, this signal decides the behavior of sign extend module. - If the ImmSrc is any bits between 3'b000 to 3'b101, the sign extend would perform extension based on Figure 50 cases. - If the ImmSrc is any bits different from 3'b000 to 3'b101, the sign extend would output 32-bits zeros.
ImmExt	Output	32	- The signal gets to be updated whenever two input signals of sign extend change their values.

To verify the Sign Extend module, its two input signals are randomly generated one thousand times on each rising clock edge. This random input generation creates a stress test environment to evaluate the module's reliability. The test is considered successful if the module accurately performs sign extension for all one thousand values across various cases.

3.2.9. Control Unit

The control signal functions as a decode unit, interpreting the opcode, funct3, and funct7 fields to identify the type of instruction being executed. It has three inputs: opcode, funct3, and funct7 (alternatively, bit 30 of the instruction can replace funct7). The control signal generates ten outputs, each controlling a specific component. Most outputs are single-bit, except for ALUCtrl, ImmSrc, and resultSrc, which are multi-bit. Typically, a control signal would produce an ALUCtrl signal that connects to an ALUop component, which then determines the ALU's operation. However, in this processor, the control unit directly manages the ALU's behavior. The control unit is implemented using a Verilog HDL 'case' block, differentiated by the opcode of each instruction type. A total of ten cases are defined: nine for the instruction formats mentioned previously and one for the NOP instruction. Within each case, a set of signals is assigned to control specific components, directing the data path accordingly.

3.3. Instruction Formats

The RV32I base instruction set architecture defines four primary instruction formats: R, I, S, and U, with two additional formats, B and J, making a total of six instruction formats. For simplicity in decoding, the register source fields (Rs1 and Rs2) and the register destination field (Rd) are consistently positioned across all formats in the RISC-V processor. Immediate values are always sign-extended, with the sign bit located at bit 31 of the instruction to optimize the sign-extension circuitry.

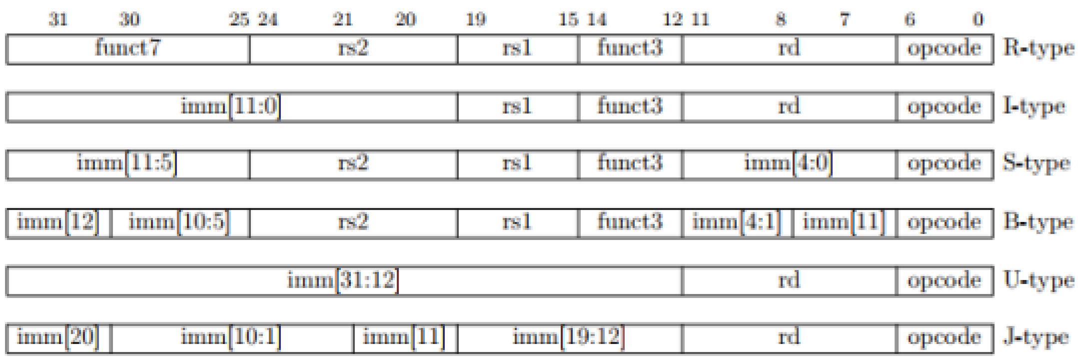


Figure 23: *RISC-V base instruction formats*

The diagram highlights the key distinctions between the S and B instruction formats in the RV32I ISA. In the B format, a 12-bit immediate field encodes branch offsets as multiples of two, where the middle bits (imm[10:1]) and the sign bit remain unchanged, and the lowest bit in the S format (instr[7]) is repurposed to encode a high-order bit in the B format, eliminating the need to shift all instruction-encoded immediate bits left by one in hardware. Conversely, the U and J formats differ in their handling of the 20-bit immediate: in the U format, the immediate is shifted left by 12 bits, while in the J format, it is shifted left by one bit. The arrangement of instruction bits in the U and J formats is optimized to maximize overlap with other formats and with each other, enhancing decoding efficiency.

3.3.1. R Format

In the RV32I ISA, all R-format instructions share the same opcode, 0110011, and are differentiated by their funct3 field and bit 30 of the instruction. Two notable exceptions are the SUB (subtract) and SRA (shift right arithmetic) instructions. The SUB instruction shares the same funct3 field as the ADD instruction, while the SRA instruction shares the same funct3 field as the SRL (shift right logical) instruction. These pairs are distinguished by bit 5 of the funct7 field, which corresponds to bit 30 of the instruction. Additionally, Figure 25 highlights the data path for R-format instructions in red, showing the flow through the processor's components.

0000000	Rs2	Rs1	000	Rd	0110011	ADD
0100000	Rs2	Rs1	000	Rd	0110011	SUB
0000000	Rs2	Rs1	001	Rd	0110011	SLL
0000000	Rs2	Rs1	010	Rd	0110011	SLT
0000000	Rs2	Rs1	011	Rd	0110011	SLTU
0000000	Rs2	Rs1	100	Rd	0110011	XOR
0000000	Rs2	Rs1	101	Rd	0110011	SRL
0100000	Rs2	Rs1	101	Rd	0110011	SRA
0000000	Rs2	Rs1	110	Rd	0110011	OR
0000000	Rs2	Rs1	111	Rd	0110011	AND

Figure 24: *R-format*

Instruction	Name	Description
ADD	Addition	Rd = Rs1 + Rs2
SUB	Subtraction	Rd = Rs1 - Rs2
SLL	Shift left logical	Rd = Rs1 << Rs2
SLT	Set less than	Rd = (Rs1 < Rs2) ? 1 : 0
SLTU	Set less than unsigned	Rd = (Rs1 < Rs2) ? 1 : 0
XOR	XOR	Rd = Rs1 ^ Rs2
SRL	Shift right logical	Rd = Rs1 >> Rs2
SRA	Shift right arithmetic	Rd = {Rs1[31]? 16’hffff:16’b0 , Rs1 >> Rs2}
OR	Or	Rd = Rs1 | Rs2
AND	And	Rd = Rs1 & Rs2

Figure 25: *R-format description*

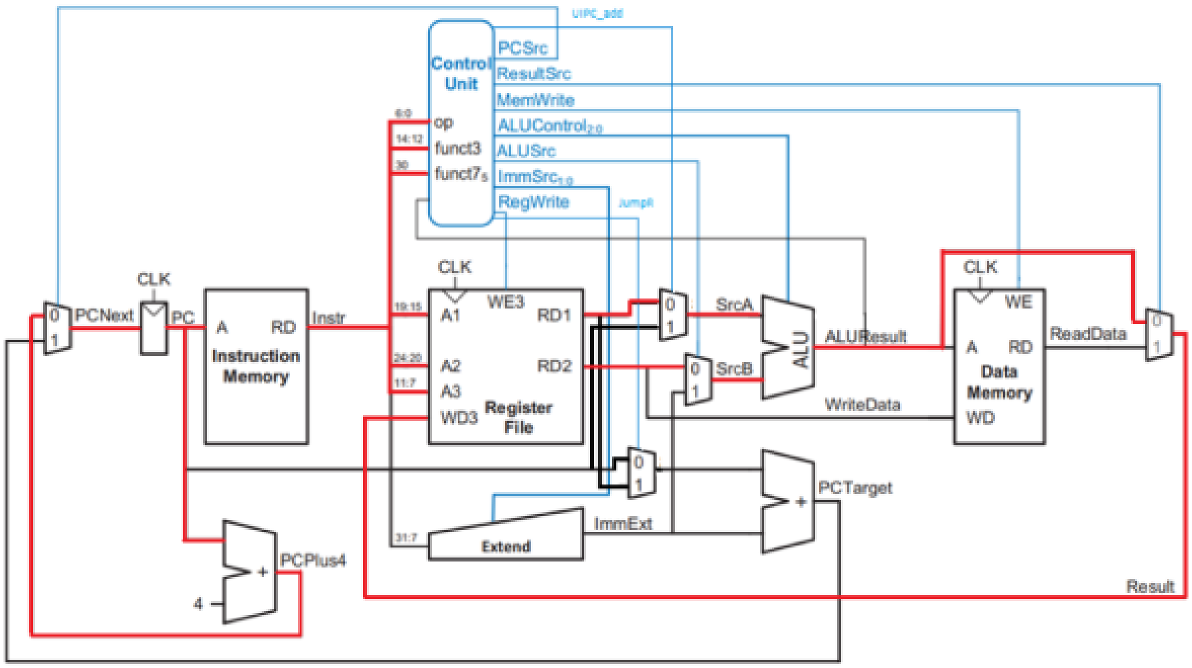


Figure 26: *R-format data path*

Figure 26 enumerates all R-type instructions within the RV32I base ISA, comprising ten unique instructions. Figure 24 details the operation of each R-type instruction, with each performing a distinct computation. Notably, the SLT (Set Less Than) and SLTU (Set Less Than Unsigned) instructions both execute comparisons but differ in their approach: SLT evaluates signed values, whereas SLTU compares unsigned values. The SRA (Shift Right Arithmetic) instruction is distinctive, performing a right shift on the source register (Rs1) and populating the destination register (Rd) such that the upper 16 bits are sign-extended, and the lower 16 bits consist of bits 15 to 0 from the shifted Rs1. Figure 26 illustrates the data path for R-type instructions, with red lines highlighting the active components involved. As R-type instructions do not involve load or store operations, they bypass the data memory (DM), rendering it unnecessary for this format.

3.3.2. I Format

Similar to the R-format, I-formats all have the same opcode, which is 0010011, and is distinct from each other by funct3 field and 30 of instructions. The figure 28 is 26 and their field off I type. The difference between I-type and R type are the sub operation, since I type can add with a negative immediate, therefore, SUBI instruction is not needed.

Imm[11:0]		Rs1	000	Rd	0010011	ADDI
Imm[11:0]		Rs1	010	Rd	0010011	SLTI
Imm[11:0]		Rs1	011	Rd	0010011	SLTIU
Imm[11:0]		Rs1	100	Rd	0010011	XORI
Imm[11:0]		Rs1	110	Rd	0010011	ORI
Imm[11:0]		Rs1	111	Rd	0010011	ANDI
0000000	Shamt	Rs1	001	Rd	0010011	SLLI
0000000	Shamt	Rs1	101	Rd	0010011	SRLI
0100000	Shamt	Rs1	101	Rd	0010011	SRAI

Figure 27: *I-format*

Instruction	Name	Description
ADDI	Addition Immediate	Rd = Rs1 + Immediate
SLTI	Set less than Immediate	Rd = (Rs1 < Immediate) ? 1 : 0
SLTIU	Set less than unsigned immediate	Rd = (Rs1 < Immediate) ? 1 : 0
XORI	XOR immediate	Rd = Rs1 ^ Immediate
ORI	OR immediate	Rd = Rs1 | Immediate
ANDI	And immediate	Rd = Rs1 & Immediate
SLLI	Shift left logical immediate	Rd = Rs1 << Immediate
SRLI	Shift right logical immediate	Rd = Rs1 >> Immediate
SRAI	Shift right arithmetic immediate	Rd = {Rs1[31]? 16'hffff:16'b0, Rs1>>Immediate}

Figure 28: *I-format description*

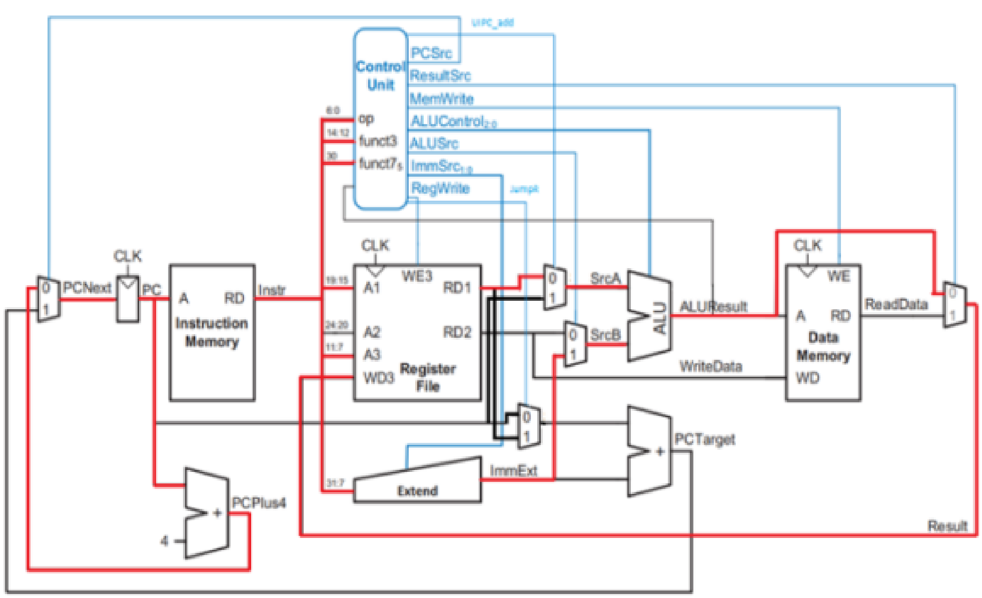


Figure 29: *I-format data path*

Unlike R-type instruction, the I-type only have nine different instructions, as in figure 28. As explain above, the SUBI is not needed in this format. The R-type need SUB instruction since it perform calculation between register, and it is difficult to determine whether the register is negative or not, hence, an instruction to do subtraction is necessary for R-format. The figure 29 is a table that describe what type of computation the instruction performs. The R-format and Iformat are similar, and their instructions, too. But unlike R-type, I format perform its computation with Rs1 and an Immediate, not Rs2. Figure 30 illustrates the data path of I-type instruction, and since this format uses Immediate instead of Rs2, the red line does not go through Rs2 port, but it goes through the sign extend to read to immediate value and uses it as source B of ALU. Other than that, since the two formats are almost identical, their data path also share the same direction.

Load instructions are also parts of the I format, but the opcode of load instruction is different from other I type instructions. Load copy a value from memory to register rd. Each load 28 instruction is distinguished by their funct3 field. The figures below demonstrate all load instructions and data path of load.

Imm[11:0]	Rs1	000	Rd	0000011	LB
Imm[11:0]	Rs1	001	Rd	0000011	LH
Imm[11:0]	Rs1	010	Rd	0000011	LW
Imm[11:0]	Rs1	100	Rd	0000011	LBU
Imm[11:0]	Rs1	101	Rd	0000011	LHU

Figure 30: *I-format load instructions*

Instruction	Name	Description
LW	Load word	Rd = M[Rs1 + Immediate] [31:0]
LB	Load byte	Rd = M[Rs1 + Immediate] [7:0]
LBU	Load byte unsigned	Rd = M[Rs1 + Immediate] [7:0]
LH	Load half word	Rd = M[Rs1 + Immediate] [15:0]
LHU	Load half word unsigned	Rd = M[Rs1 + Immediate] [15:0]

Figure 31: *I-type load description*

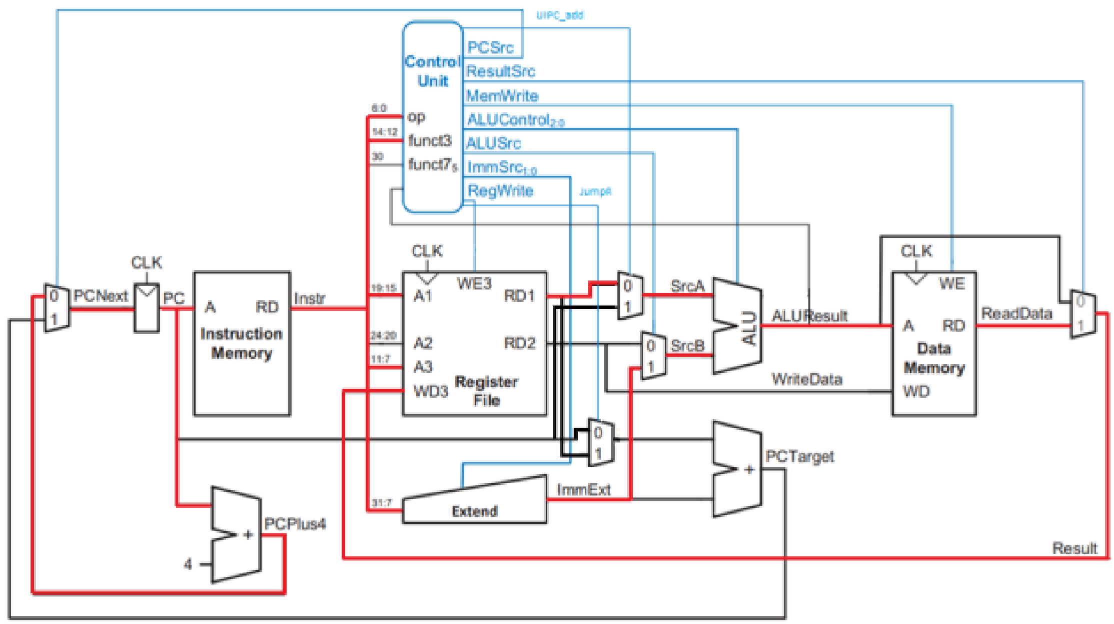


Figure 32 *: Load instruction data path*

Figure 31 have it formats the same as other I type instructions. However, I separated it from other I instruction due to the differences between the two data path, which is showcase in figure 33. Instead of ResultSrc is 0, which choose ALU output as the data to be written into Rd, the ResultSrc in this case is 1, which choose Data memory output as data to write back in the Rd. The figure 32 describe the behavior of every load instruction. For the load half word and load byte, they do not load all 32 bits value from the DM, for LH is 16 bits and 8 bits for LB. The value gets to be filled by sign extend to satisfy the 32-bit qualifications. LBU and LHU are unsigned load instructions, therefore, instead of filling the data with sign extend, the data is only filled with 0s.

Last, I-type instruction is Jump and Link Register, which is an unconditional jump, unlike branch, a conditional jump which is talked about later in this paper. The unconditional jump, like its name, do jump without comparing any value. The figures 33, 34 and 35 display format of JALR and its data path.

Imm[11:0]	Rs1	000	Rd	1100111	JALR

Figure 33*: I-type JALR instruction*

Instruction	Name	Description
JALR	Jump and link register	Rd = PC +4; PC = Rs1 + Immediate

Figure 34*: I-type JALR description*

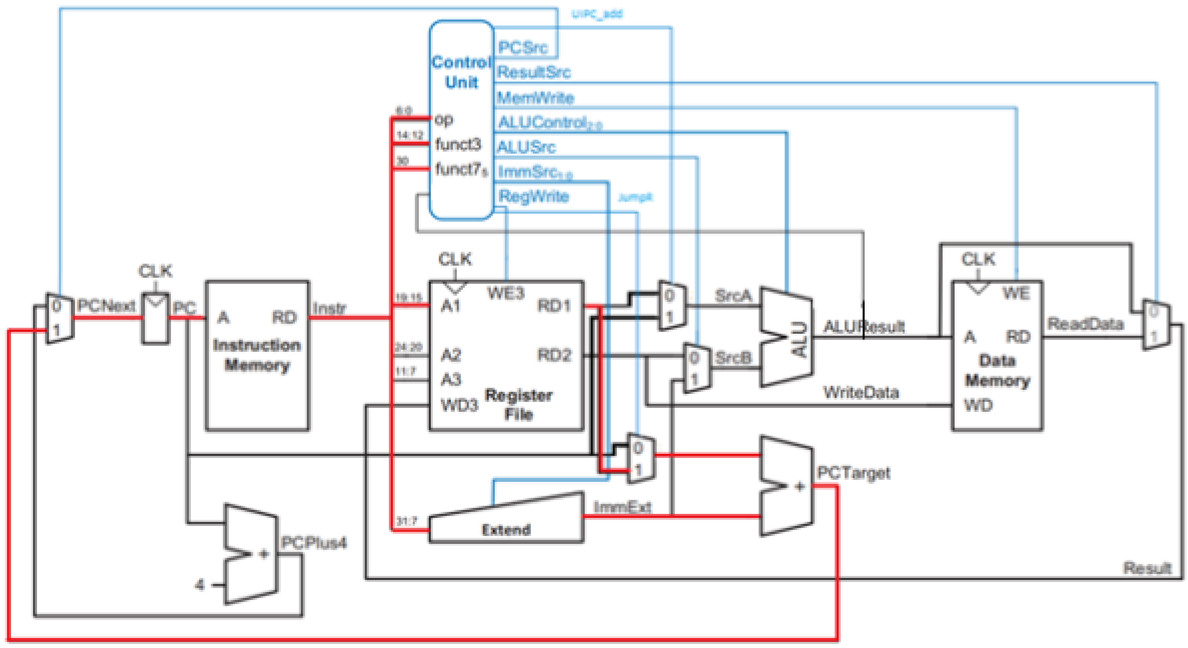


Figure 35: *JALR data path*

The figure 35 illustrate the JALR format, overall, its format is the same as any other I type instructions. However, unlike other I-type instruction, its opcode is different. This is what separate JALR, load instructions, and I-type instructions. The figure 35 display what JALR instruction does. Not only that JALR do jump to the address of the addition between Rs1 and Immediate, but also save the PC + 4 to Rd. For this instruction, the ALU is not needed, because JALR only needed to perform addition, therefore an Adder module is enough to perform the instruction.

3.3.3. S Format

S-type instruction is used to store value from register rs2 to memory. The opcode of the S type format is 0100011, and the three instructions are separated by their funct3 field. Unlike load, store needs to read two registers, rs1 for base memory address, and rs2 for data to be stored, as well as need immediate offset.

Imm[11:5]	Rs2	Rs1	000	Imm[4:0]	0100011	SB
Imm[11:5]	Rs2	Rs1	001	Imm[4:0]	0100011	SH
Imm[11:5]	Rs2	Rs1	010	Imm[4:0]	0100011	SW

Figure 36: *S-type instructions*

Instruction	Name	Description
SW	Store word	M[Rs1 + Immediate][31:0] = Rs2 [31:0]
SB	Store byte	M[Rs1 + Immediate][7:0] = Rs2 [7:0]
SH	Store half word	M[Rs1 + Immediate][15:0] = Rs2 [15:0]

Figure 37: *S-format description*

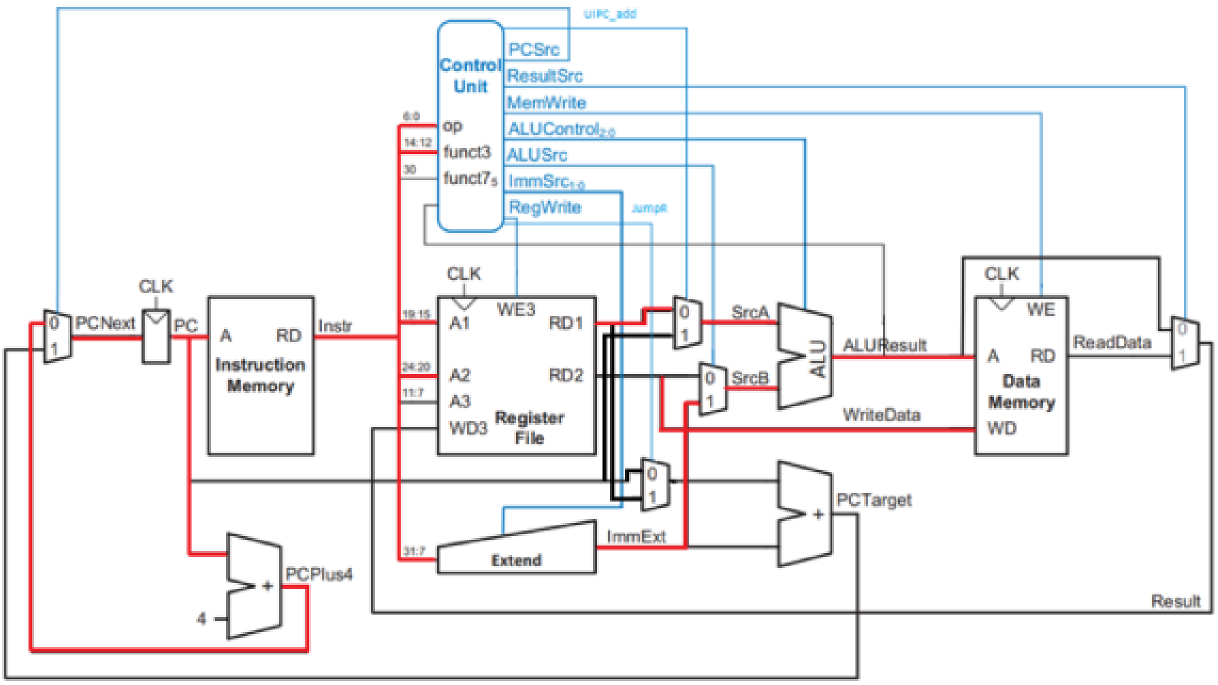


Figure 38: *S format data path*

The S-type format is used only for store instructions. Figure 36 shows the list of store instructions and figure 37 describe it behaviors. The S format only have three instructions in total, unlike load instruction, store instruction do not have unsigned instruction. The SB and SH also only store part of the data, in particular, SB only write bit 7 to bit 0 of the 32 bits values, while 32 SH write bit 15 to bit 0 of the 32 bits values, the rest of the upper bits are filled by sign extend. Figure 38 describe the direction of S format. Like the load instruction in I format, store instruction is also one of the two instruction which require DM module. However, because these instructions only store the data into DM, therefore, the data path ends at the DM.

3.3.4. U Format

U format is a solution for cases where 12-bit immediate cannot satisfy the need of programmer. So, the U format is created to input 20-bit when necessary. LUI writes the upper 20 bits of the destination with the immediate value and clears the lower 12 bits. AUIPC also write the upper 20 bits like LUI, then add the value with the current PC.

Imm[31:12]	Rd	0110111	LUI
Imm[31:12]	Rd	0010111	AUIPC

Figure 39: *U format*

Instruction	Name	Description
LUI	Load upper immediate	Rd = Immediate << 12
AUIPC	Add upper immediate to PC	Rd = PC + (Immediate <<12)

Figure 40: *U-format description*

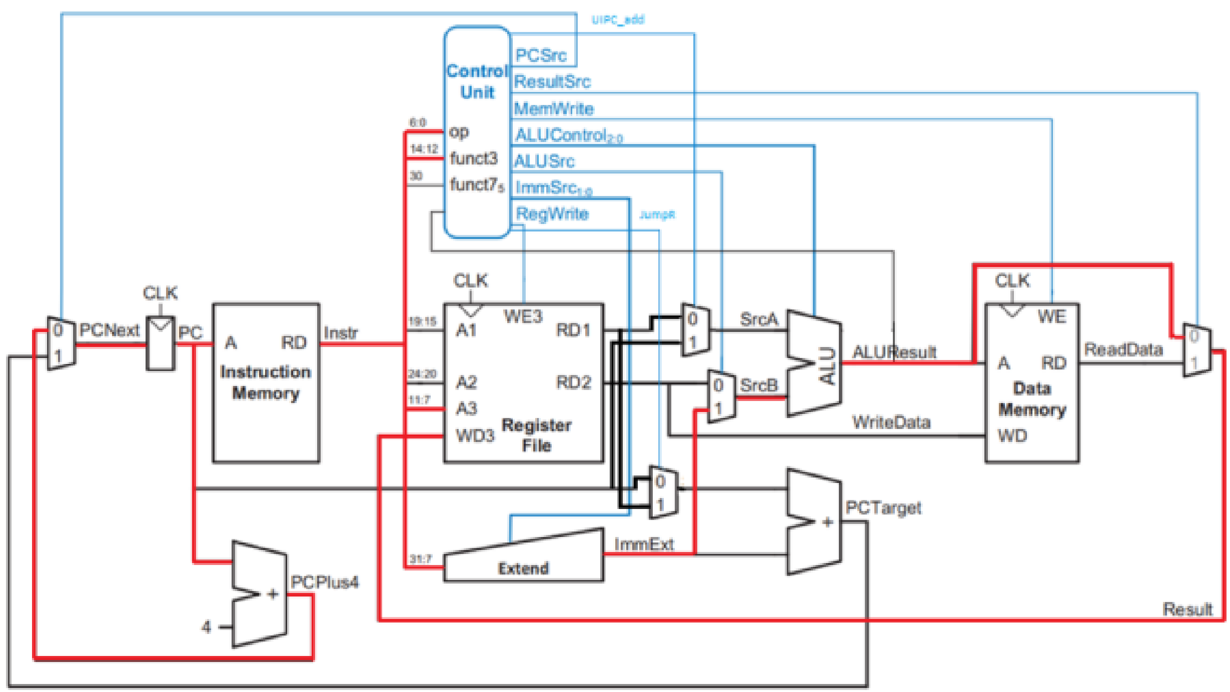


Figure 41: *LUI data path*

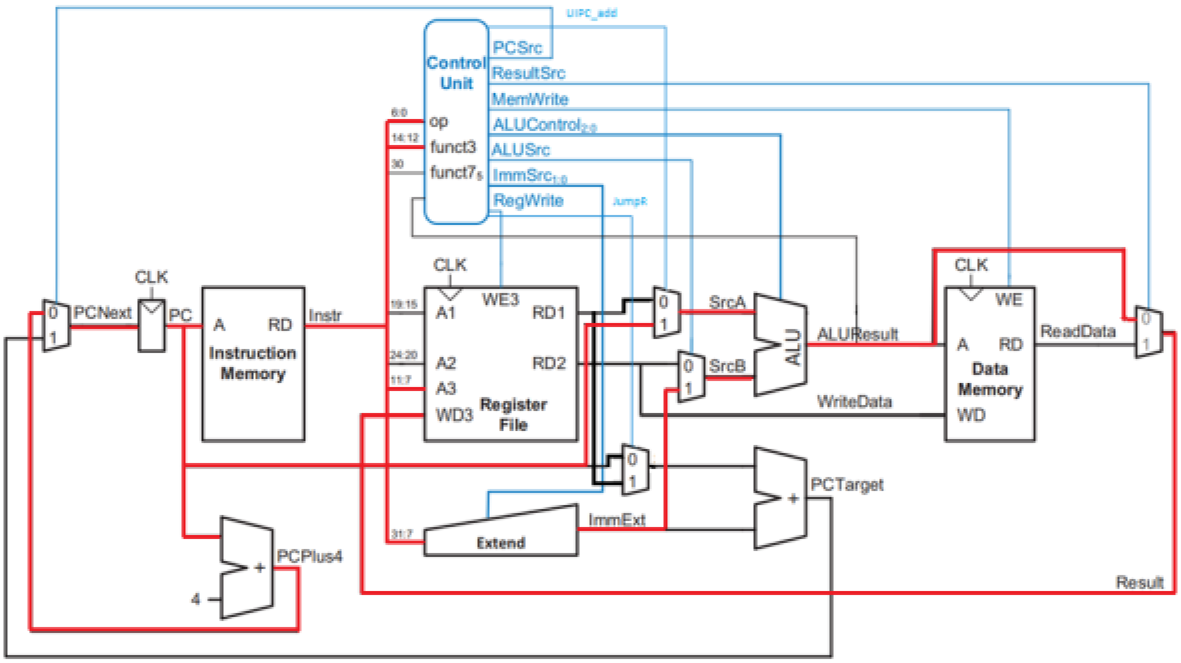


Figure 42: *AUIPC data path*

Figure 39 shows the two instructions of U type format. The two instructions have the same format, however, the opcode of the two instructions is different to distinguish the two. The next figure, which is figure 40, describes behavior of U type instructions. The LUI and AUIPC are quite the same, they both shift immediate value 12-bit to the left. The difference is that AUIPC add the shifted value with the current PC, while LUI instruction only shift the immediate. In the processor of this project, the immediate is shifted within the sign extend. The two figures 41 and 42 display the data path of LUI and AUIPC. LUI is displayed in figure 41, and since the LUI only shift immediate, therefore, the source A is not painted red. While AUIPC in figure 42 requires adding shifted value with the current PC, therefore the current PC is wired to source A of the ALU.

3.3.5. B Format

The 12-bit B immediate encodes signed offsets in multiples of 2 bytes. The offset is extended and added to the address of the branch instruction to give the target address. Branch instruction compares two registers, if the two reach the condition of the instruction, PCSrc will be set to high and enable branch.

Imm[12|10:5]	Rs2	Rs1	000	Imm[4:1|11]	1100011	BEQ
Imm[12|10:5]	Rs2	Rs1	001	Imm[4:1|11]	1100011	BNE
Imm[12|10:5]	Rs2	Rs1	100	Imm[4:1|11]	1100011	BLT
Imm[12|10:5]	Rs2	Rs1	101	Imm[4:1|11]	1100011	BGE
Imm[12|10:5]	Rs2	Rs1	110	Imm[4:1|11]	1100011	BLTU
Imm[12|10:5]	Rs2	Rs1	111	Imm[4:1|11]	1100011	BGEU

Figure 43: *B format*

Instruction	Name	Description
BEQ	Branch equal	If (Rs1 == Rs2) => PC += Immediate
BNE	Branch not equal	If (Rs1 != Rs2) => PC += Immediate
BLT	Branch less than	If (Rs1 < Rs2) => PC += Immediate
BGE	Branch greater or equal	If (Rs1 >= Rs2) => PC += Immediate
BLTU	Branch less than (U)	If (Rs1 < Rs2) => PC += Immediate
BGEU	Branch greater than or equal (U)	If (Rs1 >= Rs2) => PC += Immediate

Figure 44: *B-format description*

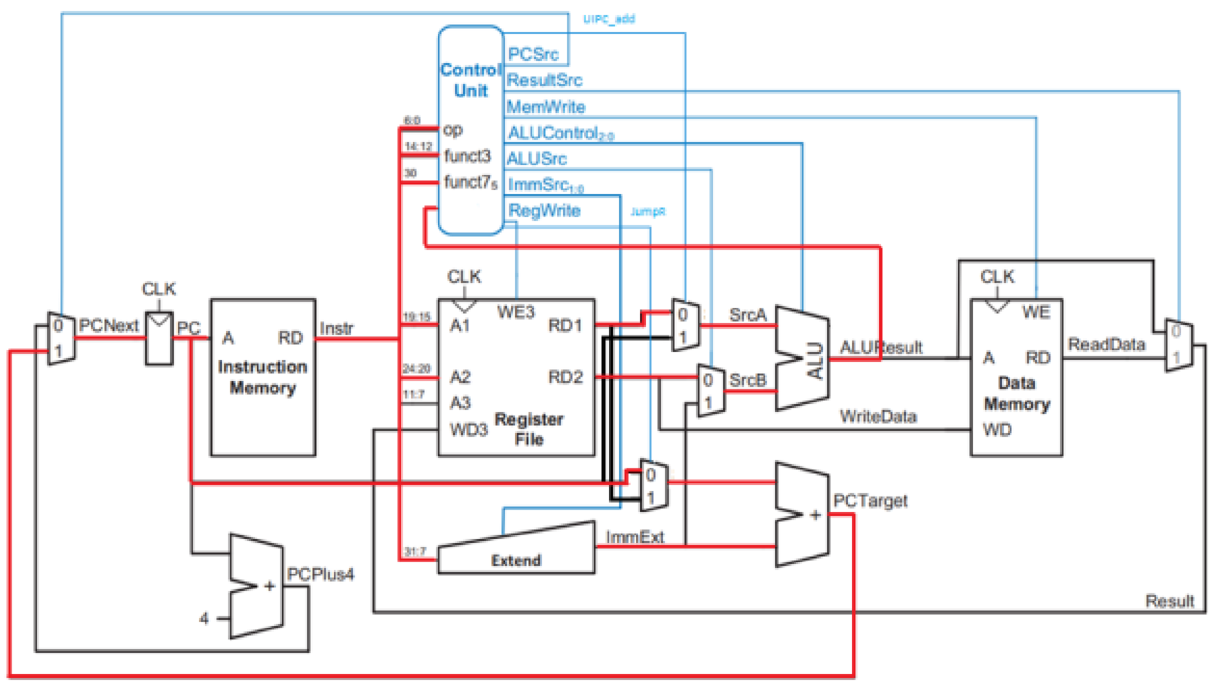


Figure 45: *B format data path*

The B format is dedicated for branch instruction, as mentioned above, is conditional instructions. The figure 43 showcases the list of B format, there are a total of six different branch instructions. B type format distinguish each other by funct3 field, and other field are the same. Immediate of B format instructions have distinctive format from other type instructions. The immediate bit is arranged as in figure 43. The format does not read bit 0 of the immediate because the instruction address is divisible for 4. In figure 44, all branch instructions are described. With respect to its name, conditional jump, the branch instruction needs to satisfy a 36 typical requirement before performing branches. The unsigned branch compares two unsigned values, unsigned comparison is only applied for BGE and BLT. BEQ and BNE do not need unsigned comparison, because the BGE also compare if two values are equal or not. Figure 45 describes the data path of the B format. Since the branch instruction performs computation within the Adder module to calculate the next instruction address. The ALU is used to do comparison. Bit 0 of ALU output is wire to the control unit to decide whether the requirement is met or not.

3.3.6. J Format

The jump and link (JAL) is the only instruction of J format, its opcode is 1101111. It saves the next program counter to the register destination Rd and jump to the destination of PC + Immediate.

Imm[20|10:1|11|19:12]	Rd	1101111	JAL

Figure 46: *J format*

Instruction	Name	Description
JAL	Jump and link	Rd = PC +4; PC += Immediate

Figure 47: *J-type description*

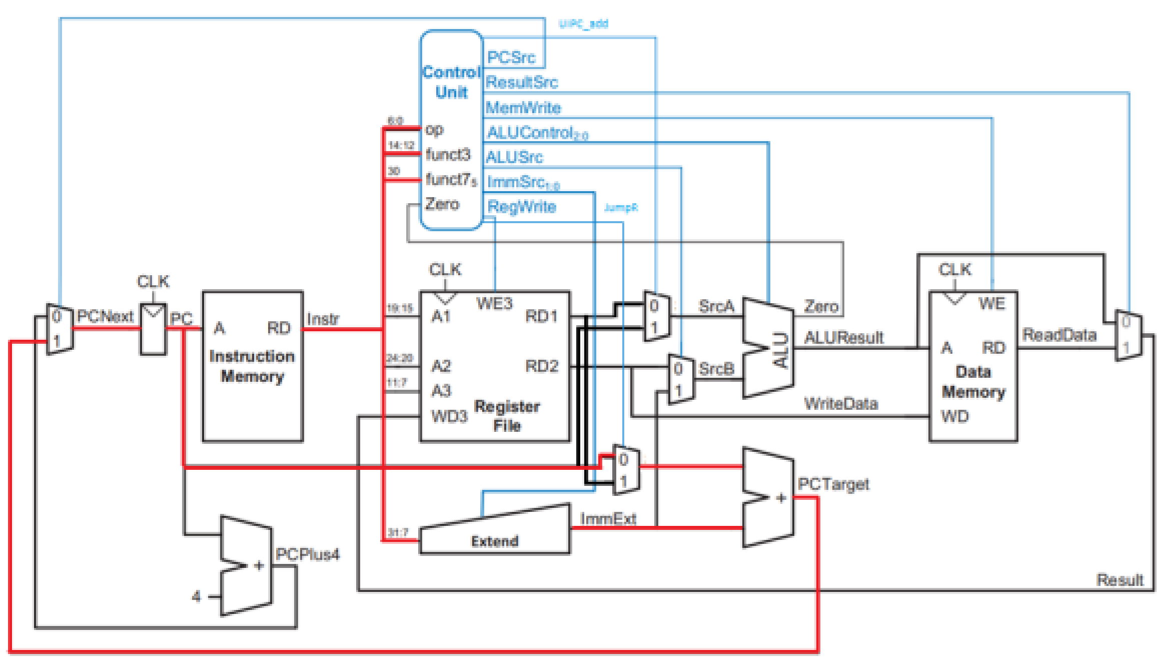


Figure 48: *J-type data path*

The J type format is used only for JAL instructions. Figure 46 illustrates the format of JAL instruction, the format does not read bit 0 of immediate due to the instruction address. Figure 47 describe the computation of JAL, it writes the next PC to Rd, at the same time calculate the jump address. This instruction can be used to jump to an instruction without having to compare any values. Figure 48 displays the data path of J type, the jump address is calculated within the Adder module, therefore, ALU is not needed.

3.4. RISC-V Component Design

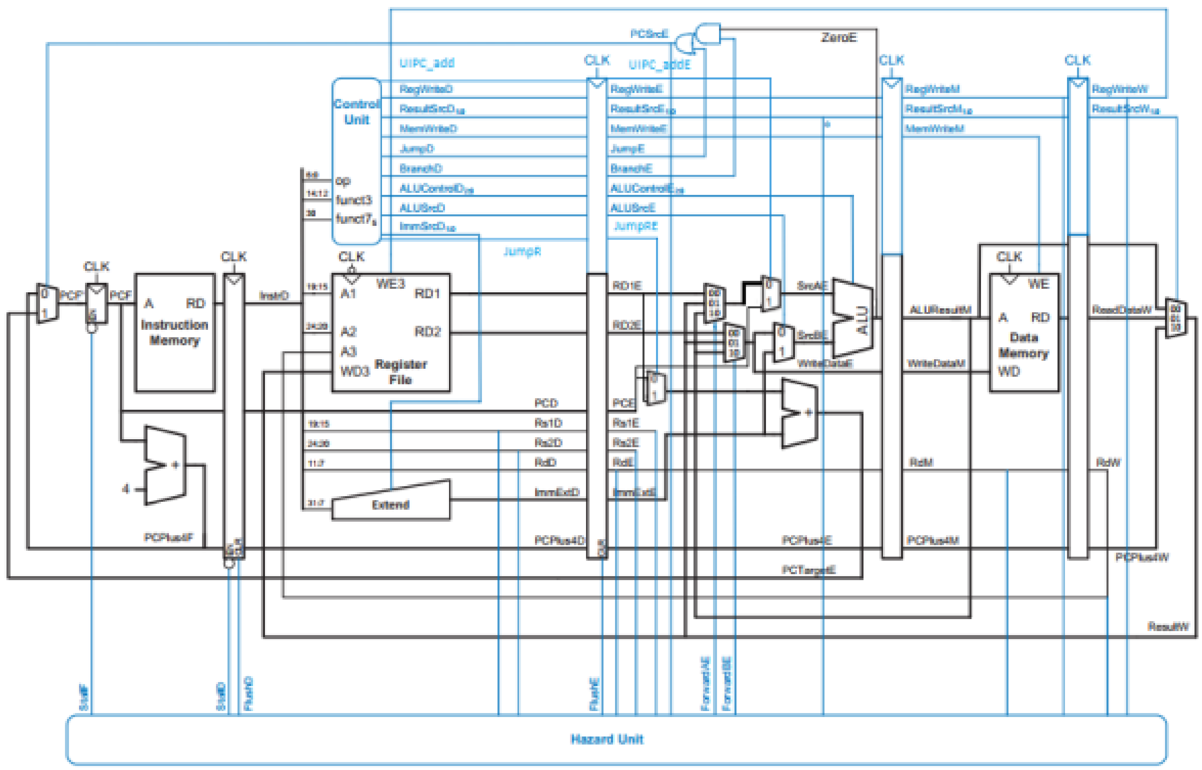


Figure 49: *Complete RISC-V processor*

CHAPTER IV

METHODOLOGY

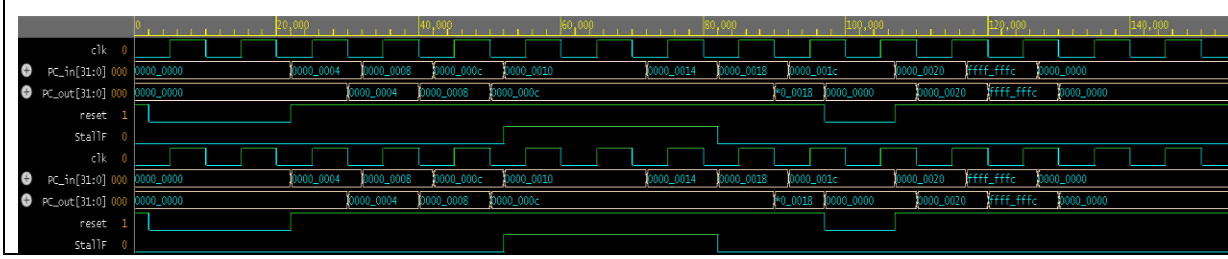
4.1. Test Module

4.1.1. Program Counter

```
module Program_Counter (clk, reset,StallF ,PC_in, PC_out);
        input clk, reset;
        input StallF;
        input [31:0] PC_in;
        output reg [31:0] PC_out;
        always @ (negedge clk or negedge reset)
        begin
                if(~reset)
                        PC_out<=0;
                else if (StallF)
                   PC_out <= PC_out;
                else
                   PC_out <= PC_in;
        end
endmodule


//Testbench
`timescale 1ns / 1ps

module tb_Program_Counter();

   reg clk;
   reg reset;
   reg StallF;
   reg [31:0] PC_in;
   wire [31:0] PC_out;


   Program_Counter uut (
      .clk(clk),
      .reset(reset),
      .StallF(StallF),
      .PC_in(PC_in),
      .PC_out(PC_out)
   );
```

```
initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

// Test cases
initial begin

    reset = 1;
    StallF = 0;
    PC_in = 32'h00000000;


    $display("Time\t\tClk\tReset\tStallF\tPC_in\t\tPC_out");
    $display("------------------------------------------------------------");


    $monitor("%0t\t\t%b\t%b\t%b\t%h\t%h",
             $time, clk, reset, StallF, PC_in, PC_out);

    // Test 1: Reset condition
    #2;
    reset = 0;  // Active low reset
    #20;

    // Test 2: Normal operation - PC increment
    reset = 1;
    PC_in = 32'h00000004;
    #10;

    PC_in = 32'h00000008;
    #10;

    PC_in = 32'h0000000C;
    #10;

    // Test 3: Stall condition
    StallF = 1;  // Enable stall
    PC_in = 32'h00000010;
    #20;

    PC_in = 32'h00000014;
    #10;

    // Test 4: Resume after stall
    StallF = 0;  // Disable stall
```

```
        PC_in = 32'h00000018;
        #10;

        // Test 5: Reset during operation
        PC_in = 32'h0000001C;
        #5;
        reset = 0;  //
        #10;

        reset = 1;
        PC_in = 32'h00000020;
        #10;

        // Test 6: Edge cases
        PC_in = 32'hFFFFFFFC;
        #10;

        PC_in = 32'h00000000;
        #10;


        #20;
        $display("\nTest completed successfully!");
        $finish;
    end

    // Dump waveform cho GTKWave
    initial begin
        $dumpfile("pc_tb.vcd");
        $dumpvars(0, tb_Program_Counter);
    end
```



Time	Clk	Reset	StallF	PC_in	PC_out
0	0	1	0	00000000	00000000
2000	0	0	0	00000000	00000000
5000	1	0	0	00000000	00000000
10000	0	0	0	00000000	00000000
15000	1	0	0	00000000	00000000
20000	0	0	0	00000000	00000000
22000	0	1	0	00000004	00000000
25000	1	1	0	00000004	00000000
30000	0	1	0	00000004	00000004
32000	0	1	0	00000008	00000004
35000	1	1	0	00000008	00000004
40000	0	1	0	00000008	00000008
42000	0	1	0	0000000c	00000008
45000	1	1	0	0000000c	00000008
50000	0	1	0	0000000c	0000000c
52000	0	1	1	00000010	0000000c
55000	1	1	1	00000010	0000000c
60000	0	1	1	00000010	0000000c
65000	1	1	1	00000010	0000000c
70000	0	1	1	00000010	0000000c
72000	0	1	1	00000014	0000000c
75000	1	1	1	00000014	0000000c
80000	0	1	1	00000014	0000000c
82000	0	1	0	00000018	0000000c
85000	1	1	0	00000018	0000000c
90000	0	1	0	00000018	00000018
92000	0	1	0	0000001c	00000018
95000	1	1	0	0000001c	00000018
97000	1	0	0	0000001c	00000000

4.1.2. Instruction Memory

```
module Instruction_Memory ( input [31:0] read_address, output [31:0] instruction);
        reg [31:0] Imemory [255:0];
        integer k;
        // I-MEM in this case is addressed by word, not by byte
```

```
        assign instruction = Imemory[read_address];
        initial
        begin
          for (k=0; k<256; k=k+1)
        begin
                        Imemory[k] = 32'b0;
                    end
                    //addi x18,x0,0x01

      Imemory[0]=32'b0000_0000_0001_00000_000_10010_0010011;
      //addi x19,x0,0x03
      Imemory[4]=32'b0000_0000_0011_00000_000_10011_0010011;
      //add x20,x20,x18
      Imemory[8]=32'b0000000_10100_10010_000_10100_0110011;
      //sw x20,0x03(x18)
      Imemory[12]=32'b0000000_10100_10010_010_00011_0100011;
      //lw x21,0x03(x18)
      Imemory[16]=32'b0000_0000_0011_10010_010_10101_0000011;
      //beq x19,x21, 0x1C
      Imemory[20]=32'b0000000_10101_10011_000_01100_1100011;
      //jalr x1,x0,0x08
      Imemory[24]=32'b0000_0000_1000_00000_000_00001_1100111;
      //lw x21,0x03(x18)
      Imemory[28]=32'b0000_0000_0011_10010_010_10101_0000011;
      //jalr x1,x0,0x20
      Imemory[32]=32'b0000_0010_0000_00000_000_00001_1100111;

    end
endmodule
//Testbench
`timescale 1ns / 1ps

module tb_Instruction_Memory();

    reg [31:0] read_address;
    wire [31:0] instruction;


    Instruction_Memory uut (
        .read_address(read_address),
        .instruction(instruction)
    );


    reg [31:0] expected_instructions [8:0];
```

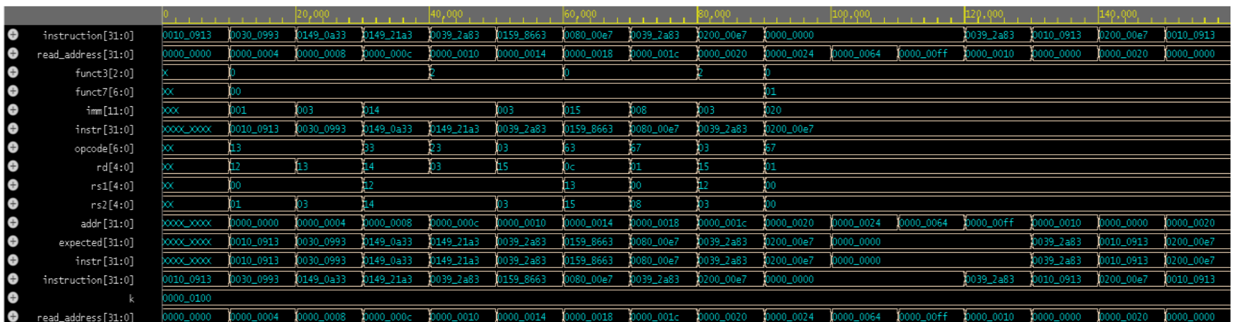
```
task display_instruction;
   input [31:0] addr;
   input [31:0] instr;
   input [31:0] expected;
   begin
      $display("Address: 0x%02X | Instruction: 0x%08X | Binary: %032b",
               addr, instr, instr);
      if (instr == expected)
         $display("✓ PASS - Instruction matches expected value");
      else
         $display("✗ FAIL - Expected: 0x%08X, Got: 0x%08X", expected, instr);
      $display("------------------------------------------------------------");
   end
endtask


task decode_instruction;
   input [31:0] instr;
   reg [6:0] opcode;
   reg [4:0] rd, rs1, rs2;
   reg [2:0] funct3;
   reg [6:0] funct7;
   reg [11:0] imm;
   begin
      opcode = instr[6:0];
      rd = instr[11:7];
      funct3 = instr[14:12];
      rs1 = instr[19:15];
      rs2 = instr[24:20];
      funct7 = instr[31:25];
      imm = instr[31:20];

      $write("Decoded: ");
      case(opcode)
         7'b0010011: begin // I-type (ADDI)
            $display("ADDI x%0d, x%0d, %0d", rd, rs1, $signed(imm));
         end
         7'b0110011: begin // R-type (ADD)
            $display("ADD x%0d, x%0d, x%0d", rd, rs1, rs2);
         end
         7'b0100011: begin // S-type (SW)
            $display("SW x%0d, %0d(x%0d)", rs2, $signed({instr[31:25], instr[11:7]}), rs1);
         end
         7'b0000011: begin // I-type (LW)
            $display("LW x%0d, %0d(x%0d)", rd, $signed(imm), rs1);
```

```
            end
            7'b1100011: begin // B-type (BEQ)
               $display("BEQ x%0d, x%0d, %0d", rs1, rs2, $signed({instr[31], instr[7],
instr[30:25], instr[11:8], 1'b0}));
            end
            7'b1100111: begin // I-type (JALR)
               $display("JALR x%0d, x%0d, %0d", rd, rs1, $signed(imm));
            end
            default: $display("Unknown instruction");
         endcase
      end
   endtask

   initial begin

      expected_instructions[0] = 32'b0000_0000_0001_00000_000_10010_0010011; // addi
x18,x0,0x01
      expected_instructions[1] = 32'b0000_0000_0011_00000_000_10011_0010011; // addi
x19,x0,0x03
      expected_instructions[2] = 32'b0000000_10100_10010_000_10100_0110011;  // add
x20,x20,x18
      expected_instructions[3] = 32'b0000000_10100_10010_010_00011_0100011;  // sw
x20,0x03(x18)
      expected_instructions[4] = 32'b0000_0000_0011_10010_010_10101_0000011; // lw
x21,0x03(x18)
      expected_instructions[5] = 32'b0000000_10101_10011_000_01100_1100011;  // beq
x19,x21, 0x1C
      expected_instructions[6] = 32'b0000_0000_1000_00000_000_00001_1100111; // jalr
x1,x0,0x08
      expected_instructions[7] = 32'b0000_0000_0011_10010_010_10101_0000011; // lw
x21,0x03(x18)
      expected_instructions[8] = 32'b0000_0010_0000_00000_000_00001_1100111; // jalr
x1,x0,0x20

      $display("=== INSTRUCTION MEMORY TEST BENCH ===");
      $display("Testing all programmed instructions...\n");


      $display("TEST 1: Reading all programmed instructions");
      $display("==============================================");

      read_address = 0;
      #10;
      display_instruction(read_address, instruction, expected_instructions[0]);
      decode_instruction(instruction);
      $display("");
```

```
read_address = 4;
#10;
display_instruction(read_address, instruction, expected_instructions[1]);
decode_instruction(instruction);
$display("");

read_address = 8;
#10;
display_instruction(read_address, instruction, expected_instructions[2]);
decode_instruction(instruction);
$display("");

read_address = 12;
#10;
display_instruction(read_address, instruction, expected_instructions[3]);
decode_instruction(instruction);
$display("");

read_address = 16;
#10;
display_instruction(read_address, instruction, expected_instructions[4]);
decode_instruction(instruction);
$display("");

read_address = 20;
#10;
display_instruction(read_address, instruction, expected_instructions[5]);
decode_instruction(instruction);
$display("");

read_address = 24;
#10;
display_instruction(read_address, instruction, expected_instructions[6]);
decode_instruction(instruction);
$display("");

read_address = 28;
#10;
display_instruction(read_address, instruction, expected_instructions[7]);
decode_instruction(instruction);
$display("");

read_address = 32;
#10;
display_instruction(read_address, instruction, expected_instructions[8]);
```

```
    decode_instruction(instruction);
    $display("");


    $display("TEST 2: Reading unprogrammed addresses (should return 0)");

$display("========================================================");

    read_address = 36;
    #10;
    display_instruction(read_address, instruction, 32'h00000000);

    read_address = 100;
    #10;
    display_instruction(read_address, instruction, 32'h00000000);

    read_address = 255;
    #10;
    display_instruction(read_address, instruction, 32'h00000000);


    $display("TEST 3: Random access test");
    $display("==========================");

    read_address = 16;
    #10;
    display_instruction(read_address, instruction, expected_instructions[4]);

    read_address = 0;
    #10;
    display_instruction(read_address, instruction, expected_instructions[0]);

    read_address = 32;
    #10;
    display_instruction(read_address, instruction, expected_instructions[8]);


    $display("TEST 4: Boundary conditions");
    $display("============================");

    read_address = 0;
    #10;
    $display("Address 0: 0x%08X", instruction);

    read_address = 254;
    #10;
```

```
        $display("Address 254: 0x%08X (should be 0)", instruction);

        $display("\n=== TEST COMPLETED ===");
        $finish;
    end

    // Dump waveform
    initial begin
        $dumpfile("imem_tb.vcd");
        $dumpvars(0, tb_Instruction_Memory);
    end

endmodule
```




```
TEST 1: Reading all programmed instructions
=============================================
Address: 0x00 | Instruction: 0x00100913 | Binary: 00000000000100000000100100010011
? PASS - Instruction matches expected value
-------------------------------------------------------------
Decoded: ADDI x18, x0, 1

Address: 0x04 | Instruction: 0x00300993 | Binary: 00000000001100000000100110010011
? PASS - Instruction matches expected value
-------------------------------------------------------------
Decoded: ADDI x19, x0, 3

Address: 0x08 | Instruction: 0x01490a33 | Binary: 00000001010010010000101000110011
? PASS - Instruction matches expected value
-------------------------------------------------------------
Decoded: ADD x20, x18, x20

Address: 0x0c | Instruction: 0x014921a3 | Binary: 00000001010010010010000110100011
? PASS - Instruction matches expected value
-------------------------------------------------------------
Decoded: SW x20, 3(x18)

Address: 0x10 | Instruction: 0x00392a83 | Binary: 00000000001110010010101010000011
? PASS - Instruction matches expected value
-------------------------------------------------------------
Decoded: LW x21, 3(x18)

Address: 0x14 | Instruction: 0x01598663 | Binary: 00000001010110011000011001100011
? PASS - Instruction matches expected value
```

```
Decoded: BEQ x19, x21, 12

Address: 0x18 | Instruction: 0x008000e7 | Binary: 00000000100000000000000011100111
? PASS - Instruction matches expected value
------------------------------------------------------------
Decoded: JALR x1, x0, 8

Address: 0x1c | Instruction: 0x00392a83 | Binary: 00000000001110010010101010000011
? PASS - Instruction matches expected value
------------------------------------------------------------
Decoded: LW x21, 3(x18)

Address: 0x20 | Instruction: 0x020000e7 | Binary: 00000010000000000000000011100111
? PASS - Instruction matches expected value
------------------------------------------------------------
Decoded: JALR x1, x0, 32

TEST 2: Reading unprogrammed addresses (should return 0)
========================================================
Address: 0x24 | Instruction: 0x00000000 | Binary: 00000000000000000000000000000000
? PASS - Instruction matches expected value
------------------------------------------------------------
Address: 0x64 | Instruction: 0x00000000 | Binary: 00000000000000000000000000000000
? PASS - Instruction matches expected value
------------------------------------------------------------
Address: 0xff | Instruction: 0x00000000 | Binary: 00000000000000000000000000000000
? PASS - Instruction matches expected value
------------------------------------------------------------
TEST 3: Random access test
==========================
Address: 0x10 | Instruction: 0x00392a83 | Binary: 00000000001110010010101010000011
? PASS - Instruction matches expected value
------------------------------------------------------------
Address: 0x00 | Instruction: 0x00100913 | Binary: 00000000000100000000100100010011
? PASS - Instruction matches expected value
------------------------------------------------------------
Address: 0x20 | Instruction: 0x020000e7 | Binary: 00000010000000000000000011100111
? PASS - Instruction matches expected value
------------------------------------------------------------
TEST 4: Boundary conditions
============================
Address 0: 0x00100913
Address 254: 0x00000000 (should be 0)

=== TEST COMPLETED ===
```

```
TEST 1: Reading all programmed instructions
==========================================
Address: 0x00 | Instruction: 0x00100913 | Binary: 00000000000100000000100100010011
? PASS - Instruction matches expected value
---------------------------------------------------------
Decoded: ADDI x18, x0, 1

Address: 0x04 | Instruction: 0x00300993 | Binary: 00000000001100000000100110010011
? PASS - Instruction matches expected value
---------------------------------------------------------
Decoded: ADDI x19, x0, 3

Address: 0x08 | Instruction: 0x01490a33 | Binary: 00000001010010010000101000110011
? PASS - Instruction matches expected value
---------------------------------------------------------
Decoded: ADD x20, x18, x20

Address: 0x0c | Instruction: 0x014921a3 | Binary: 00000001010010010010000110100011
? PASS - Instruction matches expected value
---------------------------------------------------------
Decoded: SW x20, 3(x18)

Address: 0x10 | Instruction: 0x00392a83 | Binary: 00000000001110010010101010000011
? PASS - Instruction matches expected value
---------------------------------------------------------
Decoded: LW x21, 3(x18)

Address: 0x14 | Instruction: 0x01598663 | Binary: 00000001010110011000011001100011
? PASS - Instruction matches expected value
```

```
module PC_adder (input [31:0] PC_now, output [31:0] PC_next);
   assign    PC_next = PC_now + 32'd4;
endmodule

//Testbench
`timescale 1ns / 1ps

module tb_PC_adder();
   reg [31:0] PC_now;
   wire [31:0] PC_next;

   PC_adder uut (
      .PC_now(PC_now),
      .PC_next(PC_next)
   );

   initial begin
      $display("Time\t\tPC_now\t\t\tPC_next\t\t\tExpected");
      $display("---------------------------------------------------------------");
      $monitor("%0t\t\t0x%08X\t\t0x%08X\t\t0x%08X",
               $time, PC_now, PC_next, PC_now + 32'd4);
```

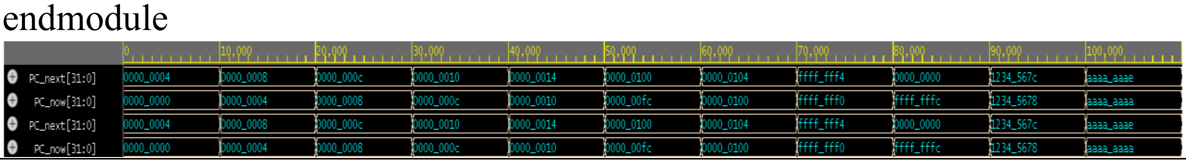
```
    PC_now = 32'h00000000;
    #10;

    PC_now = 32'h00000004;
    #10;

    PC_now = 32'h00000008;
    #10;

    PC_now = 32'h0000000C;
    #10;

    PC_now = 32'h00000010;
    #10;

    PC_now = 32'h000000FC;
    #10;

    PC_now = 32'h00000100;
    #10;

    PC_now = 32'hFFFFFFF0;
    #10;

    PC_now = 32'hFFFFFFFC;
    #10;

    PC_now = 32'h12345678;
    #10;

    PC_now = 32'hAAAAAAAA;
    #10;

    PC_now = 32'h55555554;
    #10;

    $display("\nTest completed!");
    $finish;
  end
```

```
  initial begin
    $dumpfile("pc_adder_tb.vcd");
    $dumpvars(0, tb_PC_adder);
  end

endmodule
```

4.1.3. Adder

```
module PC_adder (input [31:0] PC_now, output [31:0] PC_next);
  assign   PC_next = PC_now + 32'd4;
endmodule

//Testbench
`timescale 1ns / 1ps

module tb_PC_adder();
  reg [31:0] PC_now;
  wire [31:0] PC_next;

  PC_adder uut (
    .PC_now(PC_now),
    .PC_next(PC_next)
  );

  initial begin
    $display("Time\t\tPC_now\t\t\tPC_next\t\t\tExpected");
    $display("--------------------------------------------------------------");
    $monitor("%0t\t\t0x%08X\t\t0x%08X\t\t0x%08X",
             $time, PC_now, PC_next, PC_now + 32'd4);

    PC_now = 32'h00000000;
    #10;

    PC_now = 32'h00000004;
    #10;

    PC_now = 32'h00000008;
    #10;
```

```
    PC_now = 32'h0000000C;
    #10;

    PC_now = 32'h00000010;
    #10;

    PC_now = 32'h000000FC;
    #10;

    PC_now = 32'h00000100;
    #10;

    PC_now = 32'hFFFFFFF0;
    #10;

    PC_now = 32'hFFFFFFFC;
    #10;

    PC_now = 32'h12345678;
    #10;

    PC_now = 32'hAAAAAAAA;
    #10;

    PC_now = 32'h55555554;
    #10;

    $display("\nTest completed!");
    $finish;
  end

  initial begin
    $dumpfile("pc_adder_tb.vcd");
    $dumpvars(0, tb_PC_adder);
  end

endmodule
```

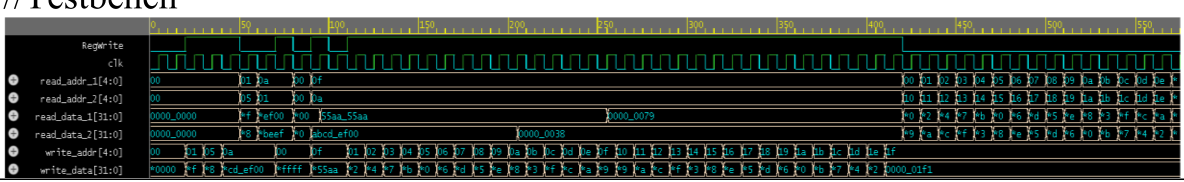


Time	PC_now	PC_next	Expected
0	0x00000000	0x00000004	0x00000004
10000	0x00000004	0x00000008	0x00000008
20000	0x00000008	0x0000000c	0x0000000c
30000	0x0000000c	0x00000010	0x00000010
40000	0x00000010	0x00000014	0x00000014
50000	0x000000fc	0x00000100	0x00000100
60000	0x00000100	0x00000104	0x00000104
70000	0xfffffff0	0xfffffff4	0xfffffff4
80000	0xfffffffc	0x00000000	0x00000000
90000	0x12345678	0x1234567c	0x1234567c
100000	0xaaaaaaaa	0xaaaaaaae	0xaaaaaaae
110000	0x55555554	0x55555558	0x55555558

4.1.4. Register File

```
module Register_File (clk,read_addr_1, read_addr_2, write_addr, read_data_1,
read_data_2, write_data, RegWrite);
        input [4:0] read_addr_1, read_addr_2, write_addr;
        input [31:0] write_data;
        input  clk,RegWrite;
        output reg [31:0] read_data_1, read_data_2;
        reg [31:0] Regfile [31:0];
        integer k;
        initial begin
        for (k=0; k<32; k=k+1)
                begin
                        Regfile[k] = 32'd0;
                end
        end

        //assign read_data_1 = Regfile[read_addr_1];
      always @( read_addr_1 or Regfile [ read_addr_1])
              begin
                if (read_addr_1 == 0) read_data_1 = 0;
                else
                begin
                read_data_1 = Regfile[read_addr_1];
```

```
//$display("read_addr_1=%d,read_data_1=%h",read_addr_1,read_data_1);
                  end
                end
        //assign read_data_2 = Regfile[read_addr_2];
      always @( read_addr_2 or Regfile [ read_addr_2])
                begin
                  if (read_addr_2 == 0) read_data_2 = 0;
                  else
                  begin
                  read_data_2 = Regfile[read_addr_2];

//$display("read_addr_2=%d,read_data_2=%d",read_addr_2,read_data_2);
                  end
                end
        always @(posedge clk)
                begin
                        if (RegWrite == 1'b1)
                          begin
                             Regfile[write_addr] = write_data;
//                           $display("write_addr=%d
write_data=%h",write_addr,write_data);
                          end
                end

endmodule

//Testbench
```



```
Time            Clk     RegWrite        WAddr   WData           RAddr1  RData1          RAddr2  RData2
---------------------------------------------------------------------------------------------------
10              1       0       0       0x00000000      0       0x00000000      0       0x00000000
20              1       0       0       0x00000000      0       0x00000000      0       0x00000000
30              1       1       1       0xdeadbeef      0       0x00000000      0       0x00000000
40              1       1       5       0x12345678      0       0x00000000      0       0x00000000
50              1       1       10      0xabcdef00      0       0x00000000      0       0x00000000
60              1       0       10      0xabcdef00      1       0xdeadbeef      5       0x12345678
70              1       0       10      0xabcdef00      10      0xabcdef00      1       0xdeadbeef
80              1       1       0       0xffffffff      10      0xabcdef00      1       0xdeadbeef
90              1       0       0       0xffffffff      0       0x00000000      0       0x00000000
100             1       1       15      0x55aa55aa      15      0x55aa55aa      10      0xabcdef00
110             1       0       15      0x55aa55aa      15      0x55aa55aa      10      0xabcdef00
x         0=0x00000000, x        16=0x00000089
x         1=0x00000002, x        17=0x0000009a
x         2=0x00000004, x        18=0x000000ac
x         3=0x00000007, x        19=0x000000bf
x         4=0x0000000b, x        20=0x000000d3
x         5=0x00000010, x        21=0x000000e8
x         6=0x00000016, x        22=0x000000fe
x         7=0x0000001d, x        23=0x00000115
x         8=0x00000025, x        24=0x0000012d
x         9=0x0000002e, x        25=0x00000146
x        10=0x00000038, x        26=0x00000160
x        11=0x00000043, x        27=0x0000017b
x        12=0x0000004f, x        28=0x00000197
x        13=0x0000005c, x        29=0x000001b4
x        14=0x0000006a, x        30=0x000001d2
x        15=0x00000079, x        31=0x000001f1
```

4.1.5. Sign Extend

```
module extend (
            input [2:0]ImmSrc,
            input [24:0] Imm,
            output reg [31:0] ImmExt);
   always @(ImmSrc or Imm)
   begin
   case (ImmSrc)
      3'b000: //lw
      begin
         ImmExt = {{20{Imm[24]}}, Imm[24:13]};
      end
      3'b001: //sw
```

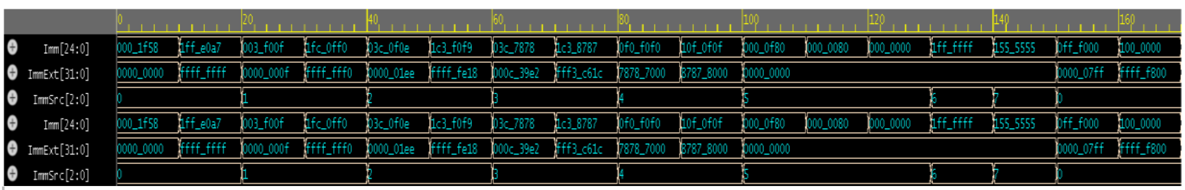
```
      begin
         ImmExt = {{20{Imm[24]}}, Imm[24:18],Imm[4:0]};
      end
      3'b010: //beq
      begin
         ImmExt =  {{20{Imm[24]}}, Imm[0], Imm[23:18], Imm[4:1], 1'b0};
      end
      3'b011: //jal
      begin
         ImmExt = {{12{Imm[24]}}, Imm[12:5], Imm[13],Imm[23:14], 1'b0};
      end
      3'b100: //LUI/AUIPC
      begin
         ImmExt = {Imm[24:5], 12'b0};
      end
      3'b101: //Shamt
      begin
         ImmExt = {27'd0,Imm[17:13]};
      end
      default:
         ImmExt = 32'd0;
   endcase
   end
endmodule

//Testbench
`timescale 1ns/1ns

module tb_extend();
   reg [2:0] ImmSrc;
   reg [24:0] Imm;
   wire [31:0] ImmExt;

   extend uut (
      .ImmSrc(ImmSrc),
      .Imm(Imm),
      .ImmExt(ImmExt)
   );

   initial begin
```

```
	$display("Time\t\tImmSrc\tImm\t\t\tImmExt\t\t\tType");
	$display("------------------------------------------------------------------------");

	// Test case 1: I-type (lw) - ImmSrc = 000
	ImmSrc = 3'b000;
	Imm = 25'b0000000000001111101011000; // Positive immediate
	#10;
	$display("%0t\t\t%b\t%b\t0x%08X\t\tI-type (lw)", $time, ImmSrc, Imm,
ImmExt);

	ImmSrc = 3'b000;
	Imm = 25'b1111111111110000010100111; // Negative immediate
	#10;
	$display("%0t\t\t%b\t%b\t0x%08X\t\tI-type (lw)", $time, ImmSrc, Imm,
ImmExt);

	// Test case 2: S-type (sw) - ImmSrc = 001
	ImmSrc = 3'b001;
	Imm = 25'b0000000111111000000001111; // Positive immediate
	#10;
	$display("%0t\t\t%b\t%b\t0x%08X\t\tS-type (sw)", $time, ImmSrc, Imm,
ImmExt);

	ImmSrc = 3'b001;
	Imm = 25'b1111111000000111111110000; // Negative immediate
	#10;
	$display("%0t\t\t%b\t%b\t0x%08X\t\tS-type (sw)", $time, ImmSrc, Imm,
ImmExt);

	// Test case 3: B-type (beq) - ImmSrc = 010
	ImmSrc = 3'b010;
	Imm = 25'b0001111000000111100001110; // Positive branch offset
	#10;
	$display("%0t\t\t%b\t%b\t0x%08X\t\tB-type (beq)", $time, ImmSrc, Imm,
ImmExt);

	ImmSrc = 3'b010;
	Imm = 25'b1110000111111000011111001; // Negative branch offset
	#10;
```

```
    $display("%0t\t\t%b\t%b\t0x%08X\t\tB-type (beq)", $time, ImmSrc, Imm,
ImmExt);

    // Test case 4: J-type (jal) - ImmSrc = 011
    ImmSrc = 3'b011;
    Imm = 25'b0001111000111100001111000; // Positive jump offset
    #10;
    $display("%0t\t\t%b\t%b\t0x%08X\t\tJ-type  (jal)",  $time,  ImmSrc,  Imm,
ImmExt);

    ImmSrc = 3'b011;
    Imm = 25'b1110000111000011110000111; // Negative jump offset
    #10;
    $display("%0t\t\t%b\t%b\t0x%08X\t\tJ-type  (jal)",  $time,  ImmSrc,  Imm,
ImmExt);

    // Test case 5: U-type (LUI/AUIPC) - ImmSrc = 100
    ImmSrc = 3'b100;
    Imm = 25'b0111100001111000011110000; // Upper immediate
    #10;
    $display("%0t\t\t%b\t%b\t0x%08X\t\tU-type (LUI)", $time, ImmSrc, Imm,
ImmExt);

    ImmSrc = 3'b100;
    Imm = 25'b1000011110000111100001111; // Upper immediate
    #10;
    $display("%0t\t\t%b\t%b\t0x%08X\t\tU-type (LUI)", $time, ImmSrc, Imm,
ImmExt);

    // Test case 6: Shift amount - ImmSrc = 101
    ImmSrc = 3'b101;
    Imm = 25'b0000000000000111110000000; // Shift amount = 15
    #10;
    $display("%0t\t\t%b\t%b\t0x%08X\t\tShamt",    $time,    ImmSrc,    Imm,
ImmExt);

    ImmSrc = 3'b101;
    Imm = 25'b0000000000000000010000000; // Shift amount = 1
    #10;
```

```
    $display("%0t\t\t%b\t%b\t0x%08X\t\tShamt", $time, ImmSrc, Imm, ImmExt);

    ImmSrc = 3'b101;
    Imm = 25'b0000000000000000000000000; // Shift amount = 0
    #10;
    $display("%0t\t\t%b\t%b\t0x%08X\t\tShamt", $time, ImmSrc, Imm, ImmExt);

    // Test case 7: Default case
    ImmSrc = 3'b110;
    Imm = 25'b1111111111111111111111111;
    #10;
    $display("%0t\t\t%b\t%b\t0x%08X\t\tDefault", $time, ImmSrc, Imm, ImmExt);

    ImmSrc = 3'b111;
    Imm = 25'b1010101010101010101010101;
    #10;
    $display("%0t\t\t%b\t%b\t0x%08X\t\tDefault", $time, ImmSrc, Imm, ImmExt);

    // Edge cases
    $display("\n--- Edge Cases ---");

    // Maximum positive I-type
    ImmSrc = 3'b000;
    Imm = 25'b0111111111111000000000000;
    #10;
    $display("%0t\t\t%b\t%b\t0x%08X\t\tI-type Max+", $time, ImmSrc, Imm, ImmExt);

    // Maximum negative I-type
    ImmSrc = 3'b000;
    Imm = 25'b1000000000000000000000000;
    #10;
    $display("%0t\t\t%b\t%b\t0x%08X\t\tI-type Max-", $time, ImmSrc, Imm, ImmExt);

    // All zeros
```

```
        ImmSrc = 3'b000;
        Imm = 25'b0000000000000000000000000;
        #10;
        $display("%0t\t\t%b\t%b\t0x%08X\t\tAll zeros", $time, ImmSrc, Imm, ImmExt);

        $display("\nTest completed!");
        $finish;
    end

    initial begin
        $dumpfile("extend_tb.vcd");
        $dumpvars(0, tb_extend);
    end

endmodule
```




```
Time            ImmSrc  Imm                        ImmExt                  Type
------------------------------------------------------------------------
10              000     0000000000001111101011000       0x00000000      I-type (lw)
20              000     1111111111110000010100111       0xffffffff      I-type (lw)
30              001     0000000111111000000001111       0x0000000f      S-type (sw)
40              001     1111111000000111111110000       0xfffffff0      S-type (sw)
50              010     0001111000000111100001110       0x000001ee      B-type (beq)
60              010     1110000111111000011111001       0xfffffe18      B-type (beq)
70              011     0001111000111100001111000       0x000c39e2      J-type (jal)
80              011     1110000111000011110000111       0xfff3c61c      J-type (jal)
90              100     0111100001111000011110000       0x78787000      U-type (LUI)
100             100     1000011110000111100001111       0x87878000      U-type (LUI)
110             101     0000000000000111110000000       0x00000000      Shamt
120             101     0000000000000000010000000       0x00000000      Shamt
130             101     0000000000000000000000000       0x00000000      Shamt
140             110     1111111111111111111111111       0x00000000      Default
150             111     1010101010101010101010101       0x00000000      Default

--- Edge Cases ---
160             000     0111111111111000000000000       0x000007ff      I-type Max+
170             000     1000000000000000000000000       0xfffff800      I-type Max-
180             000     0000000000000000000000000       0x00000000      All zeros
```

4.1.6. ALU

```
module alu
(
    // Inputs
     input  [  4:0]  alu_op_i
    ,input  [ 31:0]  alu_a_i
    ,input  [ 31:0]  alu_b_i

    // Outputs
    ,output [ 31:0]  alu_p_o
);

//-----------------------------------------------------------------
// Includes
//-----------------------------------------------------------------



//-----------------------------------------------------------------
// Registers
//-----------------------------------------------------------------
reg [31:0]      result_r;

reg [31:16]     shift_right_fill_r;
reg [31:0]      shift_right_1_r;
reg [31:0]      shift_right_2_r;
reg [31:0]      shift_right_4_r;
reg [31:0]      shift_right_8_r;

reg [31:0]      shift_left_1_r;
reg [31:0]      shift_left_2_r;
reg [31:0]      shift_left_4_r;
reg [31:0]      shift_left_8_r;

wire [31:0]     sub_res_w = alu_a_i - alu_b_i;

    //-----------------------------------------------------------------
    // greater_than_signed: Greater than operator (signed)
    // Inputs: x = left operand, y = right operand
    // Return: (int)x > (int)y
```

```
    //-----------------------------------------------------------------
    function [0:0] greater_than_signed(input  [31:0] x,y);
        reg [31:0] v;
        begin
            v = (y - x);
            if (x[31] != y[31])
                greater_than_signed = y[31];
            else
                greater_than_signed = v[31];
        end
    endfunction

//-----------------------------------------------------------------
// ALU
//-----------------------------------------------------------------
always @ (alu_op_i or alu_a_i or alu_b_i or sub_res_w)
begin
    shift_right_fill_r = 16'b0;
    shift_right_1_r = 32'b0;
    shift_right_2_r = 32'b0;
    shift_right_4_r = 32'b0;
    shift_right_8_r = 32'b0;

    shift_left_1_r = 32'b0;
    shift_left_2_r = 32'b0;
    shift_left_4_r = 32'b0;
    shift_left_8_r = 32'b0;

    case (alu_op_i)
       //----------------------------------------------
       // Shift Left
       //----------------------------------------------
       `ALU_SHIFTL :
       begin
            if (alu_b_i[0] == 1'b1)
                shift_left_1_r = {alu_a_i[30:0],1'b0};
            else
                shift_left_1_r = alu_a_i;

            if (alu_b_i[1] == 1'b1)
```

```
            shift_left_2_r = {shift_left_1_r[29:0],2'b00};
        else
            shift_left_2_r = shift_left_1_r;

        if (alu_b_i[2] == 1'b1)
            shift_left_4_r = {shift_left_2_r[27:0],4'b0000};
        else
            shift_left_4_r = shift_left_2_r;

        if (alu_b_i[3] == 1'b1)
            shift_left_8_r = {shift_left_4_r[23:0],8'b00000000};
        else
            shift_left_8_r = shift_left_4_r;

        if (alu_b_i[4] == 1'b1)
            result_r = {shift_left_8_r[15:0],16'b0000000000000000};
        else
            result_r = shift_left_8_r;
    end
    //----------------------------------------------
    // Shift Right
    //----------------------------------------------
    `ALU_SHIFTR, `ALU_SHIFTR_ARITH:
    begin
        // Arithmetic shift? Fill with 1's if MSB set
        if (alu_a_i[31] == 1'b1 && alu_op_i == `ALU_SHIFTR_ARITH)
            shift_right_fill_r = 16'b1111111111111111;
        else
            shift_right_fill_r = 16'b0000000000000000;

        if (alu_b_i[0] == 1'b1)
            shift_right_1_r = {shift_right_fill_r[31], alu_a_i[31:1]};
        else
            shift_right_1_r = alu_a_i;

        if (alu_b_i[1] == 1'b1)
            shift_right_2_r = {shift_right_fill_r[31:30], shift_right_1_r[31:2]};
        else
            shift_right_2_r = shift_right_1_r;
```

```
    if (alu_b_i[2] == 1'b1)
      shift_right_4_r = {shift_right_fill_r[31:28], shift_right_2_r[31:4]};
    else
      shift_right_4_r = shift_right_2_r;

    if (alu_b_i[3] == 1'b1)
      shift_right_8_r = {shift_right_fill_r[31:24], shift_right_4_r[31:8]};
    else
      shift_right_8_r = shift_right_4_r;

    if (alu_b_i[4] == 1'b1)
      result_r = {shift_right_fill_r[31:16], shift_right_8_r[31:16]};
    else
      result_r = shift_right_8_r;
end
//--------------------------------------------
// Arithmetic
//--------------------------------------------
`ALU_ADD :
begin
    result_r      = (alu_a_i + alu_b_i);
end
`ALU_SUB :
begin
    result_r      = sub_res_w;
end
//--------------------------------------------
// Logical
//--------------------------------------------
`ALU_AND :
begin
    result_r      = (alu_a_i & alu_b_i);
end
`ALU_OR  :
begin
    result_r      = (alu_a_i | alu_b_i);
end
`ALU_XOR :
begin
    result_r      = (alu_a_i ^ alu_b_i);
```

```
    end
    //--------------------------------------------
    // Comparision
    //--------------------------------------------
    `ALU_LESS_THAN :
    begin
        result_r      = (alu_a_i < alu_b_i) ? 32'h1 : 32'h0;
    end
    `ALU_GREATER_THAN_OR_EQUAL:
    begin
        result_r      = (alu_a_i >= alu_b_i) ? 32'h1 : 32'h0;
    end
    `ALU_LESS_THAN_SIGNED :
    begin
        if (alu_a_i[31] != alu_b_i[31])
            result_r  = alu_a_i[31] ? 32'h1 : 32'h0;
        else
            result_r  = sub_res_w[31] ? 32'h1 : 32'h0;
    end
    `ALU_GREATER_THAN_OR_EQUAL_SIGNED:
     begin
        result_r = greater_than_signed(alu_a_i,alu_b_i) | (alu_a_i == alu_b_i) ?
32'h1 : 32'h0;
     end
    `ALU_EQUAL:
    begin
        result_r = (alu_a_i == alu_b_i) ? 32'h1:32'h0;
    end
    `ALU_NOT_EQUAL:
    begin
        result_r = (alu_a_i != alu_b_i) ? 32'h1:32'h0;
    end
    //--------------------------------------------
    //Upper Imm
    //--------------------------------------------
    `ALU_LOAD_UPPER:
    begin
        result_r = alu_b_i;
     end
    default  :
```

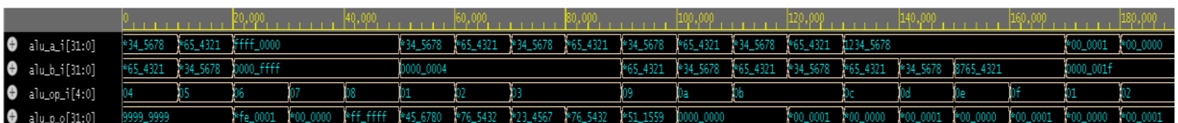
```
      begin
          result_r      = alu_a_i + alu_b_i;
      end
   endcase
end

   assign alu_p_o    = result_r;
endmodule

//Testbench
module tb_alu();

   reg [4:0]   alu_op_i;
   reg [31:0]  alu_a_i;
   reg [31:0]  alu_b_i;
   wire [31:0] alu_p_o;

   alu dut (
      .alu_op_i(alu_op_i),
      .alu_a_i(alu_a_i),
      .alu_b_i(alu_b_i),
      .alu_p_o(alu_p_o)
   );

   initial begin
      $dumpfile("alu_tb.vcd");
      $dumpvars(0, tb_alu);

      // Test ADD
      alu_op_i = `ALU_ADD;
      alu_a_i = 32'h12345678;
      alu_b_i = 32'h87654321;
      #10;
      $display("ADD: %h + %h = %h", alu_a_i, alu_b_i, alu_p_o);

      // Test SUB
      alu_op_i = `ALU_SUB;
      alu_a_i = 32'h87654321;
      alu_b_i = 32'h12345678;
      #10;
```

```
$display("SUB: %h - %h = %h", alu_a_i, alu_b_i, alu_p_o);

// Test AND
alu_op_i = `ALU_AND;
alu_a_i = 32'hFFFF0000;
alu_b_i = 32'h0000FFFF;
#10;
$display("AND: %h & %h = %h", alu_a_i, alu_b_i, alu_p_o);

// Test OR
alu_op_i = `ALU_OR;
alu_a_i = 32'hFFFF0000;
alu_b_i = 32'h0000FFFF;
#10;
$display("OR: %h | %h = %h", alu_a_i, alu_b_i, alu_p_o);

// Test XOR
alu_op_i = `ALU_XOR;
alu_a_i = 32'hFFFF0000;
alu_b_i = 32'h0000FFFF;
#10;
$display("XOR: %h ^ %h = %h", alu_a_i, alu_b_i, alu_p_o);

// Test SHIFT LEFT
alu_op_i = `ALU_SHIFTL;
alu_a_i = 32'h12345678;
alu_b_i = 32'h00000004;
#10;
$display("SHIFTL: %h << %d = %h", alu_a_i, alu_b_i, alu_p_o);

// Test SHIFT RIGHT logical
alu_op_i = `ALU_SHIFTR;
alu_a_i = 32'h87654321;
alu_b_i = 32'h00000004;
#10;
$display("SHIFTR: %h >> %d = %h", alu_a_i, alu_b_i, alu_p_o);

// Test SHIFT RIGHT arithmetic (positive)
alu_op_i = `ALU_SHIFTR_ARITH;
alu_a_i = 32'h12345678;
```

```
    alu_b_i = 32'h00000004;
    #10;
    $display("SHIFTR_ARITH (pos): %h >>> %d = %h", alu_a_i, alu_b_i,
alu_p_o);

    // Test SHIFT RIGHT arithmetic (negative)
    alu_op_i = `ALU_SHIFTR_ARITH;
    alu_a_i = 32'h87654321;
    alu_b_i = 32'h00000004;
    #10;
    $display("SHIFTR_ARITH (neg): %h >>> %d = %h", alu_a_i, alu_b_i,
alu_p_o);

    // Test LESS THAN (unsigned)
    alu_op_i = `ALU_LESS_THAN;
    alu_a_i = 32'h12345678;
    alu_b_i = 32'h87654321;
    #10;
    $display("LESS_THAN: %h < %h = %h", alu_a_i, alu_b_i, alu_p_o);

    // Test GREATER THAN OR EQUAL (unsigned)
    alu_op_i = `ALU_GREATER_THAN_OR_EQUAL;
    alu_a_i = 32'h87654321;
    alu_b_i = 32'h12345678;
    #10;
    $display("GTE: %h >= %h = %h", alu_a_i, alu_b_i, alu_p_o);

    // Test LESS THAN SIGNED (positive < negative)
    alu_op_i = `ALU_LESS_THAN_SIGNED;
    alu_a_i = 32'h12345678;
    alu_b_i = 32'h87654321;
    #10;
    $display("LESS_THAN_SIGNED: %h < %h = %h", alu_a_i, alu_b_i,
alu_p_o);

    // Test LESS THAN SIGNED (negative < positive)
    alu_op_i = `ALU_LESS_THAN_SIGNED;
    alu_a_i = 32'h87654321;
    alu_b_i = 32'h12345678;
    #10;
```

```
    $display("LESS_THAN_SIGNED: %h < %h = %h", alu_a_i, alu_b_i,
alu_p_o);

    // Test GREATER THAN OR EQUAL SIGNED
    alu_op_i = `ALU_GREATER_THAN_OR_EQUAL_SIGNED;
    alu_a_i = 32'h12345678;
    alu_b_i = 32'h87654321;
    #10;
    $display("GTE_SIGNED: %h >= %h = %h", alu_a_i, alu_b_i, alu_p_o);

    // Test EQUAL
    alu_op_i = `ALU_EQUAL;
    alu_a_i = 32'h12345678;
    alu_b_i = 32'h12345678;
    #10;
    $display("EQUAL: %h == %h = %h", alu_a_i, alu_b_i, alu_p_o);

    // Test NOT EQUAL
    alu_op_i = `ALU_NOT_EQUAL;
    alu_a_i = 32'h12345678;
    alu_b_i = 32'h87654321;
    #10;
    $display("NOT_EQUAL: %h != %h = %h", alu_a_i, alu_b_i, alu_p_o);

    // Test LOAD UPPER
    alu_op_i = `ALU_LOAD_UPPER;
    alu_a_i = 32'h12345678;
    alu_b_i = 32'h87654321;
    #10;
    $display("LOAD_UPPER: %h", alu_p_o);

    // Test edge cases for shifts
    alu_op_i = `ALU_SHIFTL;
    alu_a_i = 32'h00000001;
    alu_b_i = 32'h0000001F;
    #10;
    $display("SHIFTL max: %h << %d = %h", alu_a_i, alu_b_i, alu_p_o);

    alu_op_i = `ALU_SHIFTR;
    alu_a_i = 32'h80000000;
```

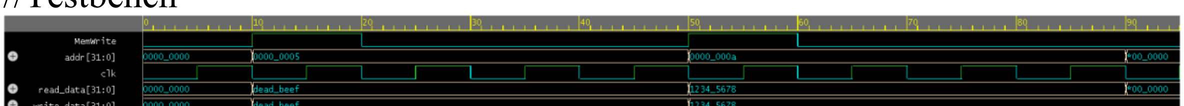
```
        alu_b_i = 32'h0000001F;
        #10;
        $display("SHIFTR max: %h >> %d = %h", alu_a_i, alu_b_i, alu_p_o);

        // Test overflow
        alu_op_i = `ALU_ADD;
        alu_a_i = 32'hFFFFFFFF;
        alu_b_i = 32'h00000001;
        #10;
        $display("ADD overflow: %h + %h = %h", alu_a_i, alu_b_i, alu_p_o);

        $finish;
    end

endmodule
```




```
ADD: 12345678 + 87654321 = 99999999
SUB: 87654321 - 12345678 = 99999999
AND: ffff0000 & 0000ffff = fffe0001
OR: ffff0000 | 0000ffff = 00000000
XOR: ffff0000 ^ 0000ffff = ffffffff
SHIFTL: 12345678 <<          4 = 23456780
SHIFTR: 87654321 >>          4 = 08765432
SHIFTR_ARITH (pos): 12345678 >>>          4 = 01234567
SHIFTR_ARITH (neg): 87654321 >>>          4 = f8765432
LESS_THAN: 12345678 < 87654321 = 95511559
GTE: 87654321 >= 12345678 = 00000000
LESS_THAN_SIGNED: 12345678 < 87654321 = 00000000
LESS_THAN_SIGNED: 87654321 < 12345678 = 00000001
GTE_SIGNED: 12345678 >= 87654321 = 00000000
EQUAL: 12345678 == 12345678 = 00000001
NOT_EQUAL: 12345678 != 87654321 = 00000000
LOAD_UPPER: 00000001
SHIFTL max: 00000001 <<         31 = 80000000
SHIFTR max: 80000000 >>         31 = 00000001
ADD overflow: ffffffff + 00000001 = 00000000
```

4.1.7. Data Memory

```
module Data_Memory (clk,addr, write_data, read_data, MemWrite);
   input [31:0] addr;
   input [31:0] write_data;
   output [31:0] read_data;
   input  MemWrite,clk;
   reg MemRead = 1;
   reg [31:0] DMemory [63:0];
   integer k;
   initial begin
      for (k=0; k<64; k=k+1)
         begin
            DMemory[k] = 32'b0;
         end
      end
   assign read_data = (MemRead) ? DMemory[addr] : 32'bx;

   always @(negedge clk)
      begin
         if (MemWrite) DMemory[addr] = write_data;
      end
endmodule

//Testbench
```




```
Time: 40 | Read Data @ addr 5 = 0xdeadbeef
Time: 80 | Read Data @ addr 10 = 0x12345678
Time: 100 | Read Data @ addr 0 = 0x00000000
```

4.1.8. Control Unit

```
module Control_Unit (
                 input [6:0]opcode,
                 input [2:0]funct3,
                 input [6:0]funct7,
```

```
            output reg UIPC_add,
            output reg JumpR,
            output reg jump,
            output reg branch,
            output reg RegWrite,
            output reg MemWrite,
            output reg ALUSrc,
            output reg [1:0]resultSrc,
            output reg [4:0]ALUCtrl,
            output reg [2:0]ImmSrc);



always @( opcode or funct3 or funct7)
begin
   case (opcode)
      7'b0110011: //R type
      begin
         RegWrite = 1;
         ImmSrc   = 3'bxxx;
         ALUSrc   = 0;
         MemWrite = 0;
         resultSrc= 2'b00;
         branch   = 0;
         jump     = 0;
         JumpR    = 0;
         UIPC_add = 0;
         if (funct7[5]==1'b1 && funct3 == 3'b000)
         begin
            $display("Sub\n");
            ALUCtrl = `ALU_SUB;
         end
         else if(funct7[5]==0 && funct3 == 3'b000)
         begin
            $display("Add\n");
            ALUCtrl = `ALU_ADD;
         end
         else if(funct7[5]==0 && funct3 == 3'b001)
         begin
            $display("SLL\n");
```

```
            ALUCtrl = `ALU_SHIFTL;
        end
        else if(funct7[5]==0 && funct3 == 3'b010)
        begin
            $display("SLT\n");
            ALUCtrl = `ALU_LESS_THAN_SIGNED;
        end
        else if(funct7[5]==0 && funct3 == 3'b011)
        begin
            $display("SLTU\n");
            ALUCtrl = `ALU_LESS_THAN;
        end
        else if(funct7[5]==0 && funct3 == 3'b100)
        begin
            $display("XOR\n");
            ALUCtrl = `ALU_XOR;
        end
        else if(funct7[5]==0 && funct3 == 3'b101)
        begin
            $display("SRL\n");
            ALUCtrl = `ALU_SHIFTR;
        end
        else if(funct7[5]==1 && funct3 == 3'b101)
        begin
            $display("SRA\n");
            ALUCtrl = `ALU_SHIFTR_ARITH;
        end
        else if(funct7[5]==0 && funct3 == 3'b110)
        begin
            $display("OR\n");
            ALUCtrl = `ALU_OR;
        end
        else if(funct7[5]==0 && funct3 == 3'b111)
        begin
            $display("AND\n");
            ALUCtrl = `ALU_AND;
        end
    end
    7'b0010011: //I type
    begin
```

```
RegWrite = 1;
ALUSrc   = 1;
MemWrite = 0;
resultSrc= 2'b00;
branch   = 0;
jump     = 0;
JumpR    = 0;
UIPC_add = 0;
if( funct3 == 3'b000)
begin
   ImmSrc   = 3'b000;
   $display("AddI\n");
   ALUCtrl = `ALU_ADD;
end
else if(funct3 == 3'b001)
begin
   ImmSrc   = 3'b101;
   $display("SLLI\n");
   ALUCtrl = `ALU_SHIFTL;
end
else if(funct3 == 3'b010)
begin
   ImmSrc   = 3'b000;
   $display("SLTI\n");
   ALUCtrl = `ALU_LESS_THAN_SIGNED;
end
else if(funct3 == 3'b011)
begin
   ImmSrc   = 3'b000;
   $display("SLTIU \n");
   ALUCtrl = `ALU_LESS_THAN;
end
else if(funct3 == 3'b100)
begin
   ImmSrc   = 3'b000;
   $display("XORI\n");
   ALUCtrl = `ALU_XOR;
end
else if(funct7[5]==0 && funct3 == 3'b101)
begin
```

```
            ImmSrc   = 3'b101;
            $display("SRLI\n");
            ALUCtrl = `ALU_SHIFTR;
        end
        else if(funct7[5]==1 && funct3 == 3'b101)
        begin
            ImmSrc   = 3'b101;
            $display("SRAI\n");
            ALUCtrl = `ALU_SHIFTR_ARITH;
        end
        else if(funct3 == 3'b110)
        begin
            ImmSrc   = 3'b000;
            $display("ORI\n");
            ALUCtrl = `ALU_OR;
        end
        else if(funct3 == 3'b111)
        begin
            ImmSrc   = 3'b000;
            $display("AND\n");
            ALUCtrl = `ALU_AND;
        end
    end
    7'b0000011: //lw
    begin
        RegWrite = 1;
        ImmSrc   = 3'b000;
        ALUSrc   = 1;
        MemWrite = 0;
        resultSrc= 2'b01;
        branch   = 0;
        jump     = 0;
        JumpR    = 0;
        UIPC_add = 0;
        ALUCtrl  = `ALU_ADD;
        $display("Load\n");
    end
    7'b0100011: //sw
    begin
        RegWrite = 0;
```

```
        ImmSrc   = 3'b001;
        ALUSrc   = 1;
        MemWrite = 1;
        resultSrc= 2'b00;
        branch   = 0;
        jump     = 0;
        JumpR    = 0;
        UIPC_add = 0;
        ALUCtrl  = `ALU_ADD;
        $display("Store\n");
    end
    7'b1100011: //branch
    begin
        RegWrite = 0;
        ImmSrc   = 3'b010;
        ALUSrc   = 0;
        MemWrite = 0;
        resultSrc= 2'bxx;
        branch   = 1;
        jump     = 0;
        JumpR    = 0;
        UIPC_add = 0;
        if (funct3 == 3'b000)
            begin
                ALUCtrl  = `ALU_EQUAL;
                $display("BEQ\n");
            end
        else if (funct3 == 3'b001)
            begin
                ALUCtrl  = `ALU_NOT_EQUAL;
                $display("BNE\n");
            end
        else if (funct3 == 3'b100)
            begin
                ALUCtrl  = `ALU_LESS_THAN_SIGNED;
                $display("BLT\n");
            end
        else if (funct3 == 3'b101)
            begin
                ALUCtrl  = `ALU_GREATER_THAN_OR_EQUAL_SIGNED;
```

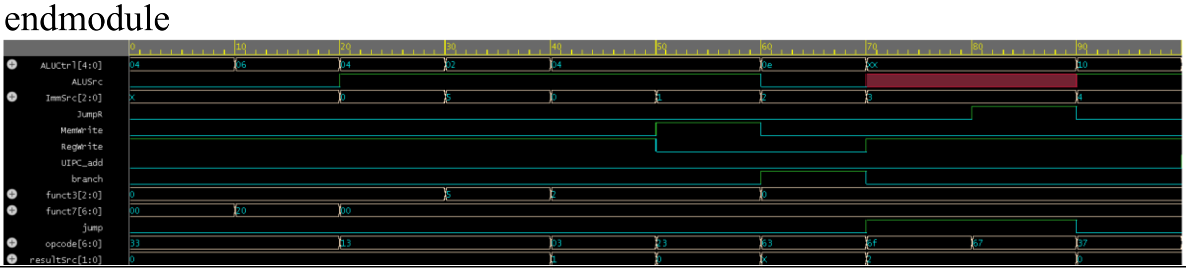
```
          $display("BGE\n");
        end
      else if (funct3 == 3'b110)
        begin
          ALUCtrl  = `ALU_LESS_THAN;
          $display("BLTU\n");
        end
      else if (funct3 == 3'b111)
        begin
          ALUCtrl  = `ALU_GREATER_THAN_OR_EQUAL;
          $display("BGEU\n");
        end
    end
    7'b1101111: //jal
    begin
      RegWrite = 1;
      ImmSrc   = 3'b011;
      ALUSrc   = 1'bx;
      MemWrite = 0;
      resultSrc= 2'b10;
      branch   = 0;
      jump     = 1;
      JumpR    = 0;
      UIPC_add = 0;
      ALUCtrl  = 5'bxxxxx;
      $display("JAL\n");
    end

    7'b1100111: //jalr
    begin
      RegWrite = 1;
      ImmSrc   = 3'b011;
      ALUSrc   = 1'bx;
      MemWrite = 0;
      resultSrc= 2'b10;
      branch   = 0;
      jump     = 1;
      JumpR    = 1;
      UIPC_add = 0;
      ALUCtrl  = 5'bxxxxx;
```

```
    $display("JALR\n");
  end

  7'b0110111: //LUI
  begin
    RegWrite = 1;
    ImmSrc   = 3'b100;
    ALUSrc   = 1;
    MemWrite = 0;
    resultSrc= 2'b00;
    branch   = 0;
    jump     = 0;
    JumpR    = 0;
    UIPC_add = 0;
    ALUCtrl  = `ALU_LOAD_UPPER;
  end

  7'b0010111: //AUIPC
  begin
    RegWrite = 1;
    ImmSrc   = 3'b100;
    ALUSrc   = 1;
    MemWrite = 0;
    resultSrc= 2'b00;
    branch   = 0;
    jump     = 0;
    JumpR    = 0;
    UIPC_add = 1;
    ALUCtrl  = `ALU_ADD;
  end

  default:
  begin
    RegWrite = 0;
    ImmSrc   = 3'b000;
    ALUSrc   = 1'b0;
    MemWrite = 0;
    resultSrc= 2'b00;
    branch   = 0;
    jump     = 0;
```

```
            JumpR    = 0;
            UIPC_add = 0;
            ALUCtrl  = `ALU_NONE;
         end
      endcase
   end
endmodule

//Testbench

module tb_Control_Unit;

   // Inputs
   reg [6:0] opcode;
   reg [2:0] funct3;
   reg [6:0] funct7;

   // Outputs
   wire UIPC_add;
   wire JumpR;
   wire jump;
   wire branch;
   wire RegWrite;
   wire MemWrite;
   wire ALUSrc;
   wire [1:0] resultSrc;
   wire [4:0] ALUCtrl;
   wire [2:0] ImmSrc;

   // Instantiate the Control Unit
   Control_Unit uut (
      .opcode(opcode),
      .funct3(funct3),
      .funct7(funct7),
      .UIPC_add(UIPC_add),
      .JumpR(JumpR),
      .jump(jump),
      .branch(branch),
      .RegWrite(RegWrite),
      .MemWrite(MemWrite),
```

```
      .ALUSrc(ALUSrc),
      .resultSrc(resultSrc),
      .ALUCtrl(ALUCtrl),
      .ImmSrc(ImmSrc)
   );

   initial begin

$display("Time\tOpcode\tFunct3\tFunct7\tALUCtrl\tRegWrite\tMemWrite\tBranch\tJump\tJun
      $display("-----------------------------------------------------------------------------------------------

      // R-type: ADD
      opcode = 7'b0110011; funct3 = 3'b000; funct7 = 7'b0000000; #10;
      display_outputs();

      // R-type: SUB
      opcode = 7'b0110011; funct3 = 3'b000; funct7 = 7'b0100000; #10;
      display_outputs();

      // I-type: ADDI
      opcode = 7'b0010011; funct3 = 3'b000; funct7 = 7'b0000000; #10;
      display_outputs();

      // I-type: SRLI
      opcode = 7'b0010011; funct3 = 3'b101; funct7 = 7'b0000000; #10;
      display_outputs();

      // Load: LW
      opcode = 7'b0000011; funct3 = 3'b010; funct7 = 7'b0000000; #10;
      display_outputs();

      // Store: SW
      opcode = 7'b0100011; funct3 = 3'b010; funct7 = 7'b0000000; #10;
      display_outputs();

      // Branch: BEQ
      opcode = 7'b1100011; funct3 = 3'b000; funct7 = 7'b0000000; #10;
      display_outputs();

      // JAL
```

```
        opcode = 7'b1101111; funct3 = 3'b000; funct7 = 7'b0000000; #10;
        display_outputs();

        // JALR
        opcode = 7'b1100111; funct3 = 3'b000; funct7 = 7'b0000000; #10;
        display_outputs();

        // LUI
        opcode = 7'b0110111; funct3 = 3'b000; funct7 = 7'b0000000; #10;
        display_outputs();

        // AUIPC
        opcode = 7'b0010111; funct3 = 3'b000; funct7 = 7'b0000000; #10;
        display_outputs();

        $display("Test completed.");
        $finish;
    end

    task display_outputs;
    begin
        $display("%0t\t%07b\t%03b\t%07b\t%05b\t%b\t\t%b\t\t%b\t%b\t%b\t%b\t%03b\t%02b\t
            $time, opcode, funct3, funct7, ALUCtrl, RegWrite, MemWrite, branch, jump, JumpR,
            ALUSrc, ImmSrc, resultSrc, UIPC_add);
    end
    endtask

    initial begin
        $dumpfile("control_unit_tb.vcd");
        $dumpvars(0, tb_Control_Unit);
    end

endmodule
```



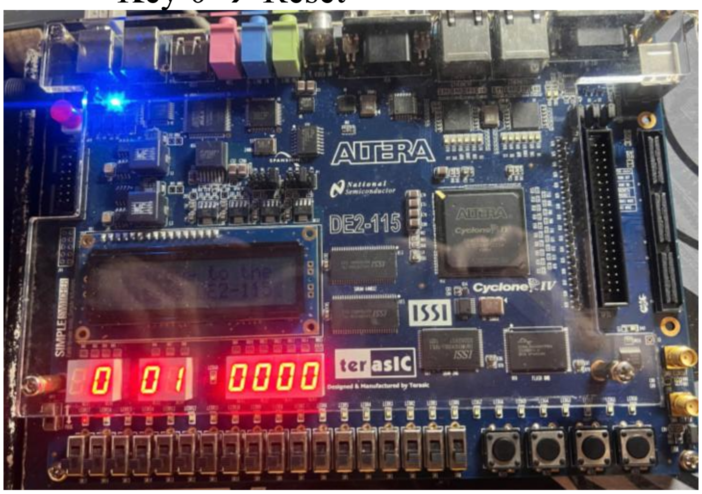
```
Time    Opcode  Funct3  Funct7  ALUCtrl RegWrite        MemWrite        Branch  Jump    JumpR   ALUSrc  ImmSrc  ResultSrc       UIPC_add
-----------------------------------------------------------------------------------------------------------------------
Add

10      0110011 000     0000000 00100   1               0               0       0       0       0       xxx     00              0
Sub

20      0110011 000     0100000 00110   1               0               0       0       0       0       xxx     00              0
AddI

30      0010011 000     0000000 00100   1               0               0       0       0       1       000     00              0
SRLI

40      0010011 101     0000000 00010   1               0               0       0       0       1       101     00              0
Load

50      0000011 010     0000000 00100   1               0               0       0       0       1       000     01              0
Store

60      0100011 010     0000000 00100   0               1               0       0       0       1       001     00              0
BEQ

70      1100011 000     0000000 01110   0               0               1       0       0       0       010     xx              0
JAL

80      1101111 000     0000000 xxxxx   1               0               0       1       0       x       011     10              0
JALR

90      1100111 000     0000000 xxxxx   1               0               0       1       1       x       011     10              0
100     0110111 000     0000000 10000   1               0               0       0       0       1       100     00              0
110     0010111 000     0000000 00100   1               0               0       0       0       1       100     00              1
```

CHAPTER V

RESULTS AND DISCUSSION

4.1. FPGA implementation

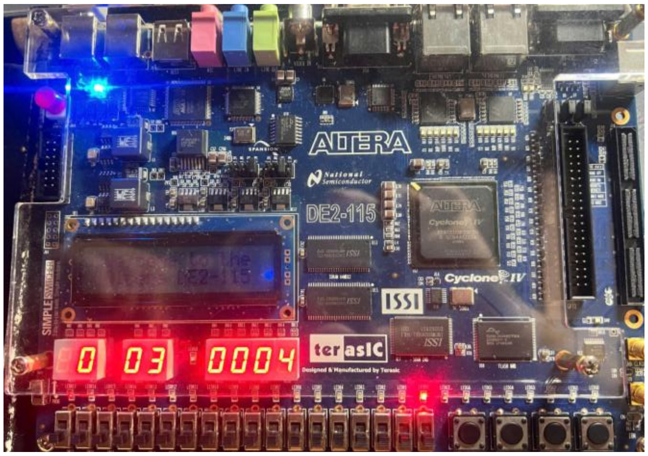
Key 0 → Reset




```
//addi x18,x0,0x01
Imemory[0]=32'b0000_0000_0001_00000_000_10010_0010011;
//addi x19,x0,0x03
Imemory[4]=32'b0000_0000_0011_00000_000_10011_0010011;
//add x20,x20,x18
Imemory[8]=32'b0000000_10100_10010_000_10100_0110011;
//sw x20,0x03(x18)
Imemory[12]=32'b0000000_10100_10010_010_00011_0100011;
//lw x21,0x03(x18)
Imemory[16]=32'b0000_0000_0011_10010_010_10101_0000011;
//beq x19,x21, 0x1C
Imemory[20]=32'b0000000_10101_10011_000_01100_1100011;
//jalr x1,x0,0x08
Imemory[24]=32'b0000_0000_1000_00000_000_00001_1100111;
//lw x21,0x03(x18)
Imemory[28]=32'b0000_0000_0011_10010_010_10101_0000011;
//jalr x1,x0,0x20
Imemory[32]=32'b0000_0010_0000_00000_000_00001_1100111;
```

PC (16 bit low)	HEX10 = 00	PC=0
PC (16 bit low)	HEX32 = 00	
ALUResult (8 bit low)	HEX54 = 00	
0 = reset, 1 = chạy	HEX76 = 00	
instruction	LEDR = 0000 0000	

Clk = 1




```
//addi x18,x0,0x01
Imemory[0]=32'b0000_0000_0001_00000_000_10010_0010011;
//addi x19,x0,0x03
Imemory[4]=32'b0000_0000_0011_00000_000_10011_0010011;
//add x20,x20,x18
Imemory[8]=32'b0000000_10100_10010_000_10100_0110011;
//sw x20,0x03(x18)
Imemory[12]=32'b0000000_10100_10010_010_00011_0100011;
//lw x21,0x03(x18)
Imemory[16]=32'b0000_0000_0011_10010_010_10101_0000011;
//beq x19,x21, 0x1C
Imemory[20]=32'b0000000_10101_10011_000_01100_1100011;
//jalr x1,x0,0x08
Imemory[24]=32'b0000_0000_1000_00000_000_00001_1100111;
//lw x21,0x03(x18)
Imemory[28]=32'b0000_0000_0011_10010_010_10101_0000011;
//jalr x1,x0,0x20
Imemory[32]=32'b0000_0010_0000_00000_000_00001_1100111;
```

PC (16 bit low) HEX10 = 04

PC=PC+4

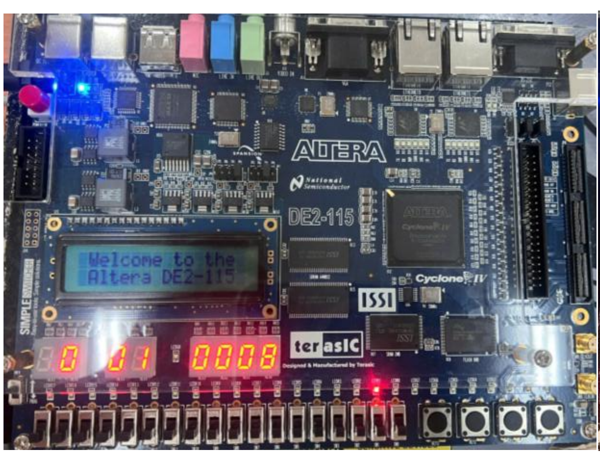
PC (16 bit low) HEX32 = 00

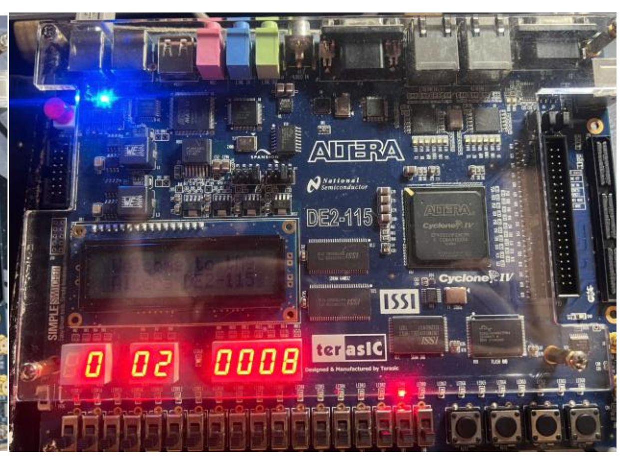
ALUResult (8 bit low) HEX54 = 03 R19=0+3=3

0 = reset, 1 = chạy HEX76 = 00

instruction index LEDR = 0000 0001 PC/4=1

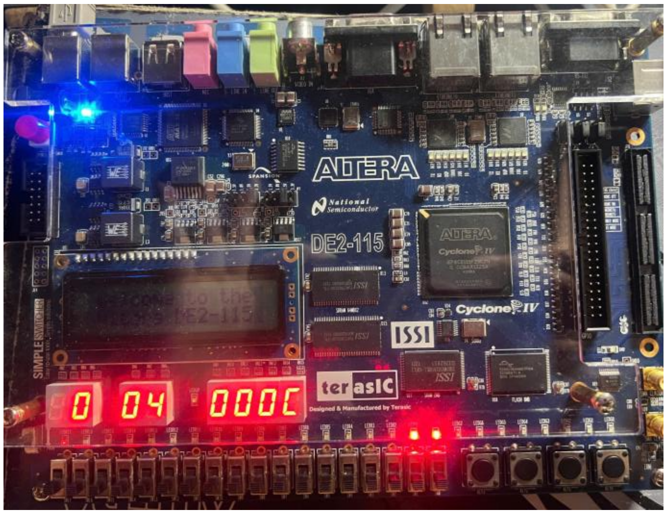
CLK=2





```
//addi x18,x0,0x01

Imemory[0]=32'b0000_0000_0001_00000_000_10010_0010011;
//addi x19,x0,0x03
Imemory[4]=32'b0000_0000_0011_00000_000_10011_0010011;
//add x20,x20,x18
Imemory[8]=32'b0000000_10100_10010_000_10100_0110011;
//sw x20,0x03(x18)
Imemory[12]=32'b0000000_10100_10010_010_00011_0100011;
//lw x21,0x03(x18)
Imemory[16]=32'b0000_0000_0011_10010_010_10101_0000011;
//beq x19,x21, 0x1c
Imemory[20]=32'b0000000_10101_10011_000_01100_1100011;
//jalr x1,x0,0x08
Imemory[24]=32'b0000_0000_1000_00000_000_00001_1100111;
//lw x21,0x03(x18)
Imemory[28]=32'b0000_0000_0011_10010_010_10101_0000011;
//jalr x1,x0,0x20
Imemory[32]=32'b0000_0010_0000_00000_000_00001_1100111;
```

PC (16 bit low)	HEX10 = 08	PC=PC+4 = 4+4=8
PC (16 bit low)	HEX32 = 00	
ALUResult (8 bit low)	HEX54 = 02	R20=0+1 next R20=R20+1=2
0 = reset, 1 = chạy	HEX76 = 00	
instruction index	LEDR = 0000 0010	PC/4=2




```
//addi x18,x0,0x01
Imemory[0]=32'b0000_0000_0001_00000_000_10010_0010011;
//addi x19,x0,0x03
Imemory[4]=32'b0000_0000_0011_00000_000_10011_0010011;
//add x20,x20,x18
Imemory[8]=32'b0000000_10100_10010_000_10100_0110011;
//sw x20,0x03(x18)
Imemory[12]=32'b0000000_10100_10010_010_00011_0100011;
//lw x21,0x03(x18)
Imemory[16]=32'b0000_0000_0011_10010_010_10101_0000011;
//beq x19,x21, 0x1C
Imemory[20]=32'b0000000_10101_10011_000_01100_1100011;
//jalr x1,x0,0x08
Imemory[24]=32'b0000_0000_1000_00000_000_00001_1100111;
//lw x21,0x03(x18)
Imemory[28]=32'b0000_0000_0011_10010_010_10101_0000011;
//jalr x1,x0,0x20
Imemory[32]=32'b0000_0010_0000_00000_000_00001_1100111;
```

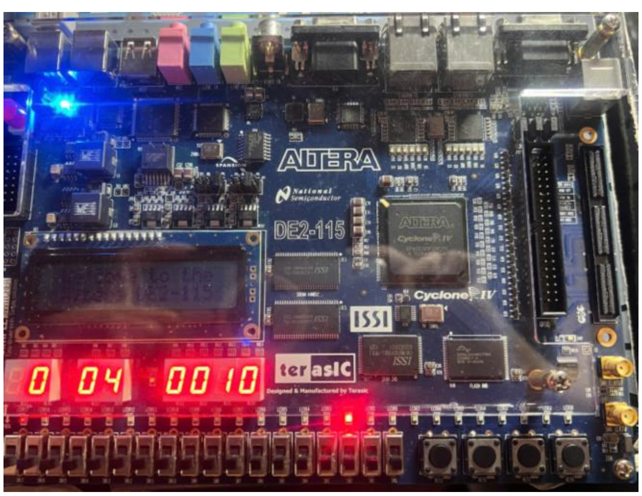
PC (16 bit low) HEX10 = 0C PC=12 (hexa decimal number)

PC (16 bit low) HEX32 = 00

ALUResult (8 bit low) HEX54 = 04 store x20=2 to **memory**[x18 + 3]=1+3=4

0 = reset, 1 = chạy HEX76 = 00

instruction index LEDR = 0000 0011 PC/4=3




```
//addi x18,x0,0x01
Imemory[0]=32'b0000_0000_0001_00000_000_10010_0010011;
//addi x19,x0,0x03
Imemory[4]=32'b0000_0000_0011_00000_000_10011_0010011;
//add x20,x20,x18
Imemory[8]=32'b0000000_10100_10010_000_10100_0110011;
//sw x20,0x03(x18)
Imemory[12]=32'b0000000_10100_10010_010_00011_0100011;
//lw x21,0x03(x18)
Imemory[16]=32'b0000_0000_0011_10010_010_10101_0000011;
//beq x19,x21, 0x1C
Imemory[20]=32'b0000000_10101_10011_000_01100_1100011;
//jalr x1,x0,0x08
Imemory[24]=32'b0000_0000_1000_00000_000_00001_1100111;
//lw x21,0x03(x18)
Imemory[28]=32'b0000_0000_0011_10010_010_10101_0000011;
//jalr x1,x0,0x20
Imemory[32]=32'b0000_0010_0000_00000_000_00001_1100111;
```

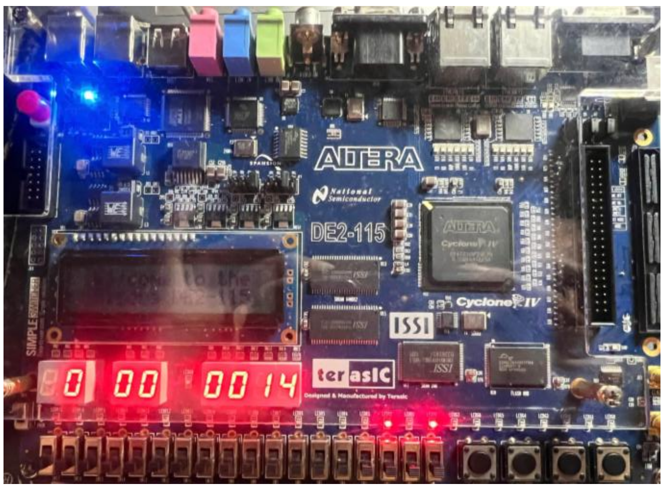
PC (16 bit low) HEX10 = 10 PC=16 (1b16 + 0)

PC (16 bit low) HEX32 = 00

ALUResult (8 bit low) HEX54 = 04 load addressMem = x18 + 3+1+3=4

0 = reset, 1 = chạy HEX76 = 00

instruction index LEDR = 0000 0100 PC/4=4




```
//addi x18,x0,0x01
Imemory[0]=32'b0000_0000_0001_00000_000_10010_0010011;
//addi x19,x0,0x03
Imemory[4]=32'b0000_0000_0011_00000_000_10011_0010011;
//add x20,x20,x18
Imemory[8]=32'b0000000_10100_10010_000_10100_0110011;
//sw x20,0x03(x18)
Imemory[12]=32'b0000000_10100_10010_010_00011_0100011;
//lw x21,0x03(x18)
Imemory[16]=32'b0000_0000_0011_10010_010_10101_0000011;
//beq x19,x21, 0x1C
Imemory[20]=32'b0000000_10101_10011_000_01100_1100011;
//jalr x1,x0,0x08
Imemory[24]=32'b0000_0000_1000_00000_000_00001_1100111;
//lw x21,0x03(x18)
Imemory[28]=32'b0000_0000_0011_10010_010_10101_0000011;
//jalr x1,x0,0x20
Imemory[32]=32'b0000_0010_0000_00000_000_00001_1100111;
```

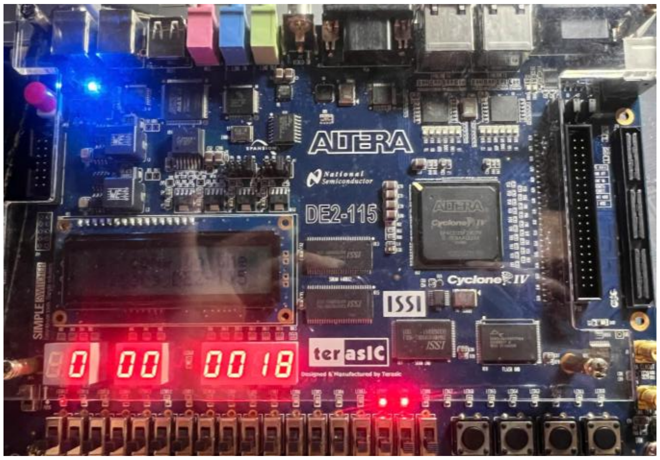
PC (16 bit low) HEX10 = 14 PC=20 (1b16 + 4)

PC (16 bit low) HEX32 = 00

ALUResult (8 bit low) HEX54 = 00 Control Unit module, x19 ≠ x21

0 = reset, 1 = chạy HEX76 = 00

instruction index LEDR = 0000 0101 PC/4=5




```
//addi x18,x0,0x01

Imemory[0]=32'b0000_0000_0001_00000_000_10010_0010011;
//addi x19,x0,0x03
Imemory[4]=32'b0000_0000_0011_00000_000_10011_0010011;
//add x20,x20,x18
Imemory[8]=32'b0000000_10100_10010_000_10100_0110011;
//sw x20,0x03(x18)
Imemory[12]=32'b0000000_10100_10010_010_00011_0100011;
//lw x21,0x03(x18)
Imemory[16]=32'b0000_0000_0011_10010_010_10101_0000011;
//beq x19,x21, 0x1C
Imemory[20]=32'b0000000_10101_10011_000_01100_1100011;
//jalr x1,x0,0x08
Imemory[24]=32'b0000_0000_1000_00000_000_00001_1100111;
//lw x21,0x03(x18)
Imemory[28]=32'b0000_0000_0011_10010_010_10101_0000011;
//jalr x1,x0,0x20
Imemory[32]=32'b0000_0010_0000_00000_000_00001_1100111;
```

PC (16 bit low) HEX10 = 18 PC=24 (1b16 + 8)

PC (16 bit low) HEX32 = 00

ALUResult (8 bit low) HEX54 = 00

0 = reset, 1 = chạy HEX76 = 00

instruction index LEDR = 0000 0110 PC/4=6


```
//addi x18,x0,0x01

Imemory[0]=32'b0000_0000_0001_00000_000_10010_0010011;
//addi x19,x0,0x03
Imemory[4]=32'b0000_0000_0011_00000_000_10011_0010011;
//add x20,x20,x18
Imemory[8]=32'b0000000_10100_10010_000_10100_0110011;
//sw x20,0x03(x18)
Imemory[12]=32'b0000000_10100_10010_010_00011_0100011;
//lw x21,0x03(x18)
Imemory[16]=32'b0000_0000_0011_10010_010_10101_0000011;
//beq x19,x21, 0x1C
Imemory[20]=32'b0000000_10101_10011_000_01100_1100011;
//jalr x1,x0,0x08
Imemory[24]=32'b0000_0000_1000_00000_000_00001_1100111;
//lw x21,0x03(x18)
Imemory[28]=32'b0000_0000_0011_10010_010_10101_0000011;
//jalr x1,x0,0x20
Imemory[32]=32'b0000_0010_0000_00000_000_00001_1100111;
```

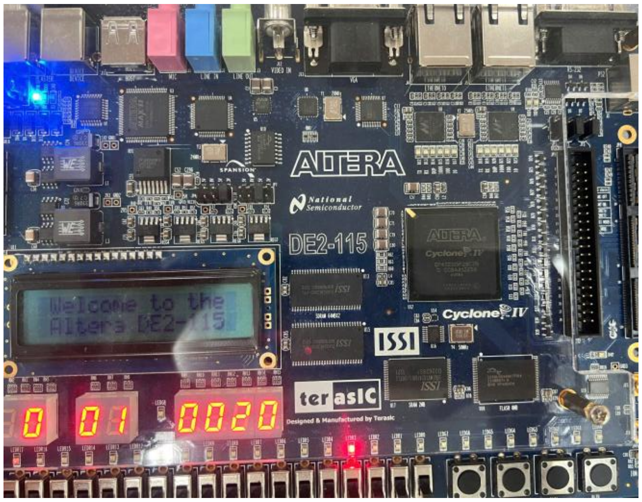
PC (16 bit low) HEX10 = 1C PC=28 (1b16 + 12)

PC (16 bit low) HEX32 = 00

ALUResult (8 bit low) HEX54 = 00

0 = reset, 1 = chạy HEX76 = 00

instruction index LEDR = 0000 0111 PC/4=7




```
//addi x18,x0,0x01
Imemory[0]=32'b0000_0000_0001_00000_000_10010_0010011;
//addi x19,x0,0x03
Imemory[4]=32'b0000_0000_0011_00000_000_10011_0010011;
//add x20,x20,x18
Imemory[8]=32'b0000000_10100_10010_000_10100_0110011;
//sw x20,0x03(x18)
Imemory[12]=32'b0000000_10100_10010_010_00011_0100011;
//lw x21,0x03(x18)
Imemory[16]=32'b0000_0000_0011_10010_010_10101_0000011;
//beq x19,x21, 0x1c
Imemory[20]=32'b0000000_10101_10011_000_01100_1100011;
//jalr x1,x0,0x08
Imemory[24]=32'b0000_0000_1000_00000_000_00001_1100111;
//lw x21,0x03(x18)
Imemory[28]=32'b0000_0000_0011_10010_010_10101_0000011;
//jalr x1,x0,0x20
Imemory[32]=32'b0000_0010_0000_00000_000_00001_1100111;
```

PC (16 bit low) HEX10 = 20 PC=32 (2b16)

PC (16 bit low) HEX32 = 00

ALUResult (8 bit low) HEX54 = 01 x19=x21

0 = reset, 1 = chạy HEX76 = 00

instruction index LEDR = 0000 1000 PC/4=8

4.2. Disscussion

In summary, the RV32I RISC-V processor successfully executes all unprivileged RV32I instructions. Several enhancements have been incorporated to support a broader range of RV32I instructions. Theoretically, this processor outperforms a single-cycle processor, but as a basic design, its performance is not yet optimized. Simulation results indicate that B-type and J-type instructions could benefit from branch prediction to address their two-cycle latency. Similarly, load instructions introduce a one-cycle latency. These instructions should be minimized to avoid degrading overall system performance. The processor's implementation of UART and I2C protocols has been validated, confirming its ability to communicate effectively via both protocols.

For future enhancements, incorporating branch prediction and multiple-issue techniques could significantly improve the processor's performance. Additionally, integrating more peripherals would enable simultaneous communication with multiple devices. The long-term objective is to develop a fully customized microprocessor.